VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

AUTOMATICKÉ TESTOVÁNÍ SYSTÉMU BEEEON

AUTOMATIC TESTING OF THE BEEEON SYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. RICHARD WOLFERT**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JAN VIKTORIN**
SUPERVISOR

BRNO 2018

Abstrakt

Táto diplomová práca sa zaoberá analýzou projektu Internetu vecí BeeeOn, návrhu a tvorbe jednotkových, integračných, systémových testov a ich automatizácii s využitím systému Kontinuálnej integrácie Jenkins CI. Teoretická časť sa venuje základom testovania softvéru a Kontinuálnej integrácii. Jadro práce tvorí špecifikácia a popis systému BeeeOn, požiadavky a potreby pre tvorbu automatických testov, ich návrhu a implementácie. V závere sa práca venuje dosiahnutým výsledkom a možnostiam rozšírenia.

Abstract

This Master's thesis is about analysis of Internet of Things project BeeeOn, design and production of unit, integration and system tests and their automation by utilization of Continuous integration system Jenkins CI. The theoretical part is devoted to software testing fundamentals and Continuous integration systems. The main point of this thesis is about specification and description of BeeeOn system, its requirements for automatic testing and its implementation. In conclusion, the results of this work and expansion possibilities are discussed.

Kľúčové slová

BeeeOn, testovanie softvéru, Kontinuálna integrácia, Internet Vecí, Jednotkové testy, Integračné testy, Systémové testy, Jenkins CI, Automatické testy.

Keywords

BeeeOn, software testing, Continuous integration, Internet of Things, Unit tests, Integration tests, System tests, Jenkins CI, Automatic testing.

Citácia

WOLFERT, Richard. *Automatické testování systému BeeeOn*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Viktorin

Automatické testování systému BeeeOn

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Ing. Jana Viktorina. Uviedol som všetky literárne pramene a publikácie z ktorých som čerpal.

. . . . . .
Richard Wolfert
23. mája 2018

Poďakovanie

Rád by som sa poďakoval vedúcemu tejto diplomovej práce Ing. Janovi Viktorinovi ako aj celému vývojovému tímu BeeeOn za poskytnuté odborné konzultácie, pomoc a pripomienky počas vypracovávania tejto práce.

Obsah

Kapitola 1

Úvod

V období intenzívnej a úzkej integrácie výpočtovej techniky a informačných systémov do rôznych častí každodenného života, či už ide o bežnú domácnosť, priemysel alebo verejnú infraštruktúru, je čoraz viac spomínaný pojem *Internet vecí* (*Internet Of Things*, *IoT*).

Internet vecí je sieť zariadení, ktoré sú integrované do rôznych oblastí života za účelom kontroly a monitorovania rôznych fyzických vlastností daného prostredia, či už ide o teplotu, vlhkosť vzduchu, intenzitu slnečného svitu, alebo mnoho ďalšieho, používajú sa pre tento účel zariadenia nazvané senzory (angl. *sensors*). Senzory sú pasívne zariadenia, schopné nejakú veličinu zmerať a v rámci IoT siete ju ďalej propagovať tak, aby sa v určitej forme dostala až k používateľovi [10]. Oproti senzorom existujú aktívne prvky, ktoré sa nazývajú aktory (*actuators*). Aktory existujú pre zaistenie druhého účelu Internetu vecí a to je fyzické ovplyvnenie nejakého stavu v reálnom prostredí. Môže ísť napríklad o otváranie alebo zatváranie okien či dverí, regulácia vykurovacieho systému, či ovládanie svetiel. Aktory na rozdiel od senzorov svoj stav nikam nepropagujú, ale dokážu naopak prijímať príkazy pre evokovanie zmeny ich stavu a v konečnom dôsledku, ovplyvnenie reálneho fyzického sveta.

V súčasnej dobe na trhu existuje veľké množstvo projektov, ktoré riešia implementáciu IoT systémov. Jedným z nich je aj projekt s názvom *BeeeOn*, ktorý vznikol na *Fakultě informačních technologií Vysokého Učení Technického v Brně*, na ktorej vznikla aj táto práca. *BeeeOn* je od ostatných systémov jedinečný v tom, že sa nesnaží ostatným systémom priamo konkurovať, naopak, snaží sa ich podporovať, integrovať a konečnému používateľovi poskytnúť jednoduché, jednotné rozhranie a prípady použitia integrovaných systémov.

V dobe písania tejto práce sa systém *BeeeOn* nachádza v štádiu intenzívneho vývoja a čím je projekt implementačne zložitejší, a robustnejší, o to náročnejšie je jeho ladenie. Softvérové chyby, ktoré sa často objavujú až pri samotnom nasadení systému majú veľmi negatívny dopad na celkový obraz projektu. Častokrát chybu odhalí vývojár, ktorý systém nasadzuje, vo väčšine prípadov sa problém nachádza v časti systému, ktorú implementoval iný vývojár. Ten musí problém preskúmať a pracne ho reportovať správnym smerom, tak aby došlo k náprave. Ladenie takéhoto problému navyše komplikuje aj fakt, že systém pracuje s fyzickými zariadeniami, ktoré nemusia byť vždy dostupné.

Vývojári teda trávia zbytočne veľa času vzájomným reportovaním a opravovaním vzniknutých problémov, čo má za následok celkové spomalenie vývoja projektu a v konečnom dôsledku dosť výrazne zvyšujú aj jeho celkovú cenu. Oveľa horšie sú však chyby, ktoré sa navonok neprejavia ani pri nasadení. Tie môžu predstavovať potencionálne bezpečnostné riziko. Nakoľko tento systém pracuje s potenciálne citlivými a osobnými údajmi používateľov, je bezpečnosť tohoto systému prvoradá. Spomenuté problémy bude možné z velkej časti eliminovať intenzívnym, štrukturovaným softvérovým testovaním.

Táto práca sa bude zaoberať návrhom, implementáciou a nasadením automatických softvérových testov pre systém *Internetu vecí BeeeOn*. Testy budú vytvorené s hlavným ohľadom na automatizáciu, jednoduchosť použitia, rozšíriteľnosť a integráciu do vývoja projektu. Hlavná motivácia tejto práce je odhalenie softvérových chýb v čo najskoršom štádiu, čím sa potencionálne zvýši celková bezpečnosť a stabilita, urýchli sa jeho vývoj, a zaistí sa určitá garancia kvality implementovaného systému

Kapitola 2

Testovanie softvéru

Prv, než sa dostanem k popisu systému BeeeOn a návrhu testov chcem venovať jednu kratšiu kapitolu samotnému softvérovému testovaniu. Vysvetlím, o čo sa vlastne jedná, akým spôsobom sa rozdeľuje a aké techniky sa pri ňom používajú. Pri spracovaní tejto kapitoly som vychádzal prednostne z publikácie *Testování softwaru* od Rona Pattona [13].

2.1 Motivácia a ciele testovania

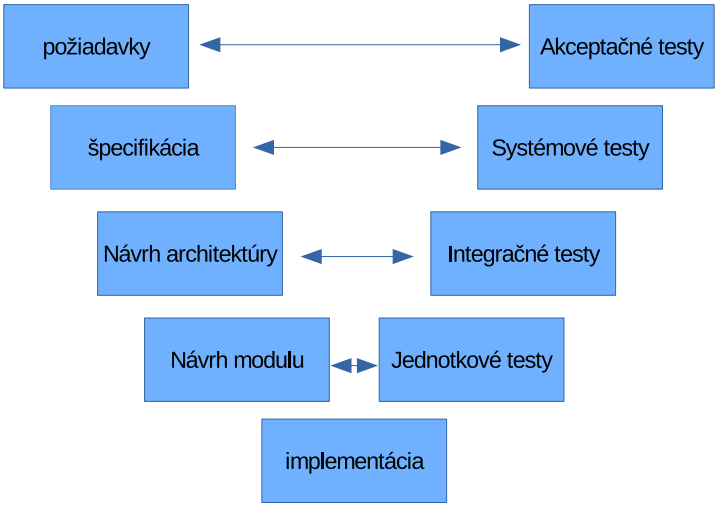
Testovanie softvéru je technická disciplína, ktorej účelom je verifikácia a validácia testovaného softvéru. Verifikácia je proces, ktorého cieľom je potvrdiť, že implementovaný softvér vyhovuje zadanej špecifikácii. Cieľom validácie je zistenie, že zadaná špecifikácia vyhovuje potrebám koncového používateľa [12]. Pre vznik softvérovej chyby musí byť splnená jedna z *nasledujúcich* podmienok:

- Softvér nerobí niečo, čo by podľa specifikácie robiť mal.
- Softvér robí niečo, čo by podľa špecifikácie robiť nemal.
- Softvér robí niečo, o čom špecifikácia nehovorí.
- Softvér nerobí niečo, o čom špecifikácia nehovorí, ale mala by hovoriť.
- Softvér je náročný na pochopenie, ťažko sa s ním pracuje, je pomalý alebo ho koncový používateľ nebude považovať za správny.

Kľúčové je pre softvérového testera vyhľadávať chyby čo najskôr, pokiaľ je to možné. Čím neskôr sa daná chyba odhalí, tým väčší bude jej dopad, cena opravy a v konečnom dôsledku celková cena projektu.

2.2 Testovacie fázy

V knihe *Softwarové inženýrství* od Iana Sommerville [17] rozdeľuje testovanie do štyroch hierarchicky zoradených fáz: jednotkové testy, integračné testy, systémové testy a akceptačné testy. Takéto rozdelenie je aj súčasťou vývojového V-modelu 2.1 [8]. V tejto podkapitole budú popísané jednotlivé úrovne.



Obr. 2.1: Vývojový V-model.

2.2.1 Jednotkové testy

V jednotkových testoch sa analyzuje najmenšia testovateľná softvérová časť (jednotka/modul). Táto časť je izolovaná od zvyšku systému a skúma sa na nej špecifikovaná množina vstupov a očakávaných výstupov. Cieľom je verifikovať, že táto implementovaná časť odpovedá tomu ako bola navrhnutá. Pre projekty písané v objektovo orientovaných jazykoch sa typicky jedná o metódu patriacu nejakej triede.

2.2.2 Integračné testy

V ďalšej testovacej úrovni, po jednotkových testoch, sa spojí niekoľko samostatných jednotiek do funkčného subsystému. Cieľom je opäť verifikovať, že jednotlivé časti integrované do väčšieho celku odpovedajú tomu ako boli špecifikované a navrhnuté. Testuje sa tu izolovaná väčšia časť celého systému a analyzuje sa, či komponenty, s ktorých je časť integrovaná medzi sebou správne komunikujú.

2.2.3 Systémové testy

Pri testovaní systému sú spojené jednotlivé subsystémy, ktoré sa testujú ako celok. Testuje sa, či sú dané komponenty navzájom kompatibilné a správne medzi sebou interagujú. Pri takejto integrácii sa prejavuje emergentné chovanie, ktoré znamená, že niektoré prvky systémových funkcií je možné pozorovať až po kombinácii daných komponentov. V tejto fázy môže dochádzať k integrácii častí, ktoré vyvinuli rôzny členovia týmu, alebo rôzne skupiny.

2.2.4 Akceptačné testy

Akceptačné testovanie je proces verifikácie a zároveň validácie. Systém v tomto prípade pracuje s ostrými a skutočnými dátami na rozdiel od predchádzajúcich fáz, kde sa používajú umelo vytvorené dáta. Táto fáza sa často nazýva aj alfa testovanie. Testuje sa tu, či produkt pracuje správne v reálnom nasadení a spĺňa používateľské požiadavky[16].

2.3 Členenie na základe povahy testov

Táto podkapitola popisuje rozdelenie testov na základe ich povahy. Takýmto spôsobom je testovanie rozdelené aj v knihe *Testování softwaru*.

2.3.1 Dynamické testovanie čiernej skrinky

Pre dynamické testovanie je špecifické to, že sa implementovaný kód spúšťa a sleduje sa jeho chovanie. Testovanie čiernej skrinky znamená, že pri testovaní nepoznáme internú štruktúru testovaného programu ani spôsob jeho práce. Pre definované vstupy porovnávame očakávané výsledky, pri čom nevidíme, čo sa deje vo vnútri testovaného softvéru. Softvér testujeme tak, ako by ho používal koncovný používateľ.

Pre testovanie softvéru poznáme dva základné spôsoby: testy splnením (*test-to-pass*) a testy zlyhaním (*test-to-fail*). V prvom prípade testujeme základnú funkčnosť programu, používame valídne vstupy pre otestovanie základných, najjednoduchších prípadov použitia. V druhom prípade používame hraničné až nevalídne vstupy a snažíme sa docieliť zlyhanie programu, čím odhalíme chyby [13].

2.3.2 Dynamické testovanie bielej skrinky

Rovnako ako predchádzajúci prípad aj v tomto prípade ide o dynamické testovanie, implementovaný kód sa teda spúšta a analyzuje za jeho behu. V tomto prípade však máme informácie o tom, ako program interne funguje, ako je štukturovaný a ako jednotlivé komponenty medzi sebou komunikujú. To nám umožňuje cielene testovať jednotlivé časti a procedúry programu [13].

2.3.3 Statické testovanie bielej skrinky

Pri statickom testovaní sa softvér nespúšta. Namiesto toho sa reviduje návrh, architektúra a samotný kód programu. Tento spôsob testovania má veľký prínos pre odhalovanie chýb pri ich vzniku. Hlavným nástrojom statického testovania bielej skrinky je revízia kódu [13].

Revízia kódu

Pod pojmom revízia kódu (*code review*) je presne definovaný proces, v rámci ktorého prebieha statické testovanie bielej skrinky. Tento proces vykonávajú ľudia, typicky členovia týmu, preto ho nie je možné plne automatizovať. Tento proces môžeme rozdeliť na 4 prvky [13]:

- Identifikácia problému – Cieľom revízie je hľadať problémy a nedostaky v analyzovanom kóde. Kritika sa musí vecne týkať samotného kódu, nie osoby, ktorá ho vytvorila.
- Dodržiavanie pravidiel – Pri revízii je dôležité dodržiavať predom dohodnutú špecifikovanú množinu pravidiel, ktoré určujú, akým spôsobom sa bude kód revidovať a čo od revízie očakávať.
- Príprava – Pred vykonaním revízie je potrebné aby každá zúčastnená strana bola pripravená svojím spôsobom do revízie prispieť. Jednotlivé zúčastnené strany môžu mať v revízii svoje role z ktorých vyplývajú jednotlivé zodpovednosti.

- Písomná správa – Výsledkom revízie je vytvorenie zápisu, ktorý je ďalej predaný vývojovému tímu.

Revízia kódu má množstvo ďalších pozitívnych vplyvov na samotný vývoj projektu, ako je napríklad rozvoj menej skúsených vývojárov od skúsenejších kolegov v rámci procesu revízie kódu. Ďalšou nespornou výhodou je zvýšenie kvality samotného kódu, nakoľko sú vývojári motivovaní po sebe svoj kód viackrát skontrolovať než ho uznajú za vhodný na revíziu.

Nevýhoda revízie je, že je v praxi dosť podceňovaná a má zdanlivo vysokú časovú réžiu [13]. Zdanlivo, pretože čas, ktorý sa stratí pri revízii kódu je potom ušetrený neskôr na problémoch, ktoré sú vyriešené v tomto procese. Riešiť tieto problémy neskôr by malo vyššiu cenu.

Statická analýza kódu

Oproti revízii existujú aj automatizované nástroje pre statické testovanie bielej skrinky. Tieto nástroje analyzujú zdrojový kód, pričom ho nespúštajú. Prispievajú k celkovej čistote a bezpečnosti kódu. Vyhľadávajú potencinálne problémy, ako napríklad nežiadúce prekrývanie rozsahu premenných, potencionálne úniky pamäte, delenie nulou, nepoužívané atribúty, metódy triedy a podobne. Výhoda týchto nástrojov je možnosť úplnej automatizácie. Existuje veľké množstvo otvorených aj proprietárnych nástrojov pre takmer všetky rozšírené programovacie jazyky. Nevýhoda týchto nástrojov sú takzvané *false positives*, kedy je kód nesprávne vyhodnotený ako problematický, alebo dokonca, chybový. Z tohoto dôvodu musia byť výsledky vždy prešetrené vývojárom.

2.3.4 Statické testovanie čiernej skrinky

Táto metóda sa nazýva aj testovanie špecifikácie. Netestuje sa žiadny bežiaci kód, preto je považovaná za statickú. Špecifikácia je výsledok, ktorý vznikol v závislosti na viacerých zdrojoch, požiadaviek zákazníka a štúdií uskutočniteľnosti. Pri testovaní výsledného dokumentu sa ho snažíme preskúmať a zistiť rôzne potencionálne problémy.

2.4 Automatizácia testov

Vývoj softvéru je dynamická činnosť, počas ktorej sa opakovane strieda proces typicky zložený z implementácie kódu, testovania a opravy nájdených chýb. Z pohľadu testovania to znamená, že navrhnuté testy budú musieť byť vykonané opakovane. Vykonávaním opakovaných testov overujeme, či boli opravené chyby z predchádzajúcej iterácie a či do implementácie neboli zanesené ďalšie, nové chyby. Takéto testovanie sa nazýva *regresné testovanie*. K tomuto účelu sa používajú nástroje na automatické testovanie. Tieto nástroje môžeme rozdeliť na *neinvazívne* a *invazívne*. *Neinvazívne* nástroje testovaný softvér skúmajú a monitorujú. *Invazívne* nástroje oproti tomu dokážu modifikovať kód programu alebo manipulovať s jeho prostredím. Je preto veľmi žiaduce používať čo najmenej invazívnych protriedkov, pretože modifikáciou samotného programu je možné docieliť skreslené výsledky testovania. Nástroje pre automatické testovanie sa vyznačujú týmito dôležitými vlastnosťami:

- Rýchlosť – Oproti manuálnemu testovaniu je možné pomocou automatizácie zrýchliť priebeh celej testovacej sady až tisíce násobne.
- Efektivita – Skracovaním času priebehu celej testovacej sady ostáva viac času pre autora testov na pokrytie viacerých prípadov použitia testovaného systému.

- Správnosť a presnosť – V prípade manuálneho testovania stoviek testovacích scenárov tester stráca pozornosť a je náchylný k chybám. Nástroje pre automatické testovanie v tomto ohľade dokážu fungovať dokonale.
- Neúnavnosť – Použité testovacie nástroje nikdy predčasne neprerušia svoju prácu z dôvodu únavy.

Automatizácia testovania má však aj svoje nevýhody. Jednou z nich je priebežne meniaca sa špecifikácia, ktorá môže znehodnotit výsledky testovacej sady, ktorá s týmto nepočíta a z tohoto dôvodu je potrebné ju priebežne aktualizovať.

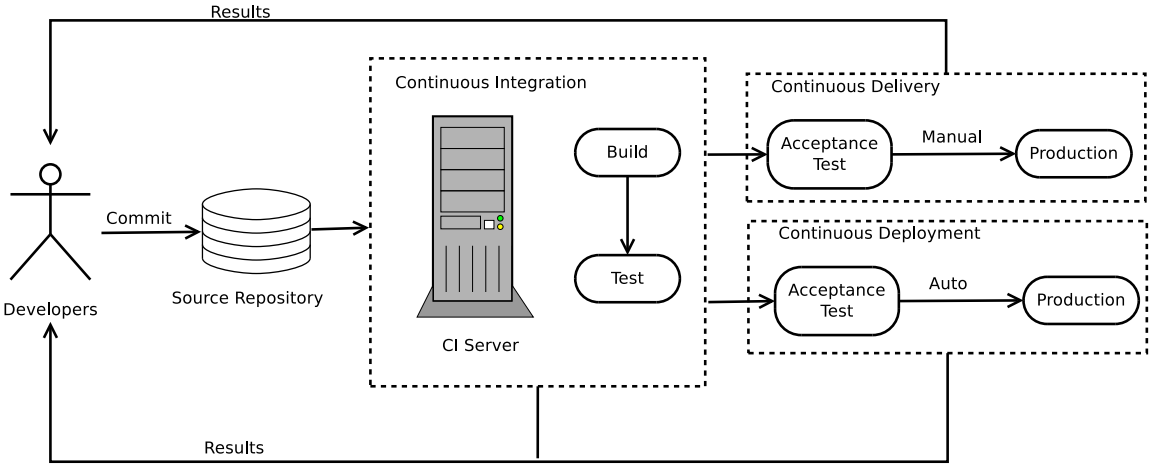
2.4.1 Systémy kontinuálnej integrácie

Kontinuálna integrácia (*Continuous Integration, CI*) je široko rozšírená praktika vývoja softvéru, pri ktorej členovia vývojového tímu veľmi často zlučujú a integrujú vyvýjaný systém. Napomáha skracovať dobu medzi vydaniami softvéru, skracuje vývojové iterácie, zvyšuje celkovú produktivitu vývojového tímu a kvalitu vyvýjaného softvéru. Tento proces v sebe zahŕňa automatickú kompiláciu a testovanie [15].

S pojmom Kontinuálna integrácia súvisia pojmy Kontinuálne doručenie (*Continuous DElivery, CDE*) a Kontinuálne nasadenie (*Continuous Deployment, CD*). V oboch prípadoch sa jedná o prirodzené rozšírenie systému Kontinuálnej integrácie. Vzťah medzi týmito pojmami je znázornený na obrázku 2.2.

Kontinuálne doručenie zabezpečuje, aby bol systém po úspešnom splnení automatických testov a akceptačných testov ihneď v stave pripravenom na produkciu. Výhodou je zníženie rizík z nasadenia a rýchlejšia odozva od používateľa systému. V prípade Kontinuálneho nasadenia je systém kontinuálne a automaticky nasadzovaný rovno do používateľského prostredia pri každej zmene.

Hlavným rysom týchto systémov je použitie zreťazenej linky(*pipeline*). Zreťazená linka je celková postupná cesta v celom procese. Celá procedúra je evokovaná pridaním nového zdrojového kódu do systému (*commit*), následne sa vyvolá automatický preklad, testovanie a v konečnom dôsledku, pri systémoch Kontinuálneho nasadenia aj samotné nasadenie systému do používateľského prostrednia.



Obr. 2.2: Vzťah medzi pojmami Kontinuálna integrácia, Kontinuálne doručenie a Kontinuálne nasadenie[15]

2.5 Limity softérového testovania

Softvérové testovanie má svoje teoretické limity. Softvérovým testom je možné dokázať prítomnosť chýb v programe, nie je ním možné dokázať, že v ňom chyby nie sú. Aby to bolo možné, museli by sme verifikovať celú množinu vstupov aplikácie, ktorá je, pri komplexnejších aplikáciách, prakticky nekonečná. Problém nájdenia všetkých chýb v programe je všeobecne nerozhodnuteľný [7].

2.6 Python unittest

Rozšíreným nástrojom pre tvorbu softvérových testov je štandardná knižnica skriptovacieho jazyka Python *unittest* [6]. V tejto podkapitole by som chcel priblížiť základné koncepty a princípy tvorby testov, ktoré táto knižnica ponúka. Dokumentácia knižnice definuje niekoľko základných pojmov, ktoré sú tiež rozšírené aj v iných systémoch pre tvorbu softvérových testov [6]:

- *test fixture*: Predstavuje inicializáciu a prípravu nutnú pre beh testov a súčasne rôzne čistiace akcie, uvolňovanie alokovaných prostriedkov a podobne. Môže sa tu nachádzať vytvorenie dočasnej databázy, spustenie potrebných procesov, vytvorenie adresárov a súborov.

- *test case*: Jedná sa o samostatnú jednotku pre testovanie. Prebieha tu samotné testovanie softvéru, kedy typicky podrobujeme program nejakou sadou vstupov a kontrolujeme očakávané výstupy.

- *test suite*: Kolekcia testov (*test case*), ktoré spolu nejakým spôsobom súvisia a mali by byť testované spolu.

- *test runner*: Táto komponenta slúži pre riadenie vykonávania samotných testov. Jej zodpovednosť je aj informovať používateľa o priebehu a poskytovať výsledky. Pre poskytnutie výsledkov môže používať grafické rozhranie, textové rozhranie alebo špeciálnu návratovú hodnotu, ktorá indikuje výsledok prebehnutých testov. Tak isto sa môže jednať o nejaký automatizovane spracovateľný formát ako je *XML*, *TAP* a podobne.

Knižnica je zameraná objektovo-orientovane a poskytuje triedu *unittest.TestCase*, ktorá tvorí základnú triedu pre všetky implementované testovacie prípady. Vlastný testovací prípad vytvoríme zdedením tejto triedy a implementovaním metód s prefixom „test“. Volitelne môžeme implementovať metódy s názvom `setUp()` a `tearDown()`. Tieto metódy predstavujú implementáciu konceptu *test fixture* a sú volané pred samotným testom, respektíve po ňom. Pre samotné vyhodnocovanie priebehu testu sú použité metódy `assert`. Knižnica poskytuje celú množinu takýchto metód, príkladom sú `assertEqual(a,b)` pre ľubovolné porovnanie dvoch hodnôt, `assertRaises(exception, callback)`, kedy očakávame vygenerovanie špecifickej výnimky, `assertAlmostEqual(a,b,precision)` pre porovnávanie čísiel s pohyblivou rádovou čiarkov a mnohé ďalšie. Ukážku použitia je možné vidieť na príklade testov hypotetickej matematickej knižnice 2.1.

```
import unittest
import math

class TestExample(unittest.TestCase):
        def setUp(self):
                math.initialise()

        def tearDown(self):
                math.cleanup()

        def test1_add(self):
                self.assertEqual(math.add(2,2), 4)

        def test2_sub(self):
                self.assertEqual(math.sub(3,1), 3)

        def test3_sqrt(self):
                self.assertAlmostEqual(math.sqrt(25), 5, 7)

        def test4_div(self):
                self.assertRaises(ZeroDivisionError, math.div(5,0))
```

Výpis 2.1: Príklad testu s využitím knižnice *Python.unittest*

Kapitola 3

BeeeOn systém

Systém BeeeOn vznikol ako *open source* a *open hardware* univerzitný projekt na Fakulte informačních technologií Vysokého učení technického v Brně, kde aj vznikla táto práca. Cieľom tohoto projektu je vytovrenie systému *Internetu vecí*, ktorý bude zaisťovať jednoduchú použiteľnosť, bezpečnosť a flexibilitu pre bežného používateľa. Systém je zameraný primárne na správu a kontrolu prostredia v rámci bežnej domácnosti, sekundárne ho bude možné využiť v administratívnych budovách alebo kanceláriách. Súčasťou projektu je aj vlastné hardvérové riešenie niektorých senzorov. Hlavnou snahou však nie je konkurovať ostatným systémom *Internetu vecí*, ale naopak, ich podpora a integrácia do jednotného systému. V konečnom dôsledku je používateľovi ponúknuté jednotné rozhranie. Pre to aby bolo možné rozumne navrhnúť testovaciu sadu pre tento systém, ktorá by pokryla základné aj rozšírené prípady použitia, potrebujeme najskôr poznať špecifikáciu a architektúru systému. Práve tomu sa venuje táto kapitola.

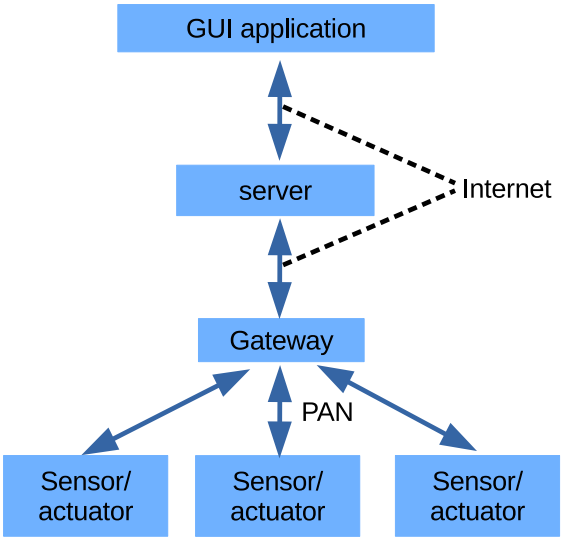
Definícia softvérovej chyby z kapitoly 2 je závislá na samotnej špecifikácii softvéru, preto je dôležité ju popísať. Nakoľko je systém BeeeOn z pohľadu vývoja dosť flexibilný, prakticky neexistuje formálne spísaná a priebežne udržiavaná aktuálna špecifikácia tohoto systému. Literatúra s takýmto prípadom počíta a tvrdí, že špecifikáciou môže byť aj samotný kód [13]. Keďže je kód projektu otvorený, verejný a na jeho dokumentáciu je pri vývoji kladený veľký dôraz, rozhodol som sa ho považovať za špecifikáciu, ktorá bude slúžiť pre podklad testovania. Pri spracovávaní tejto kapitoly som vychádzal zo samotných zdrojových kódov projektu [1], konzultácií členov a vedúcich vývojového tímu.

3.1 Architektúra systému

Ako je možné vidieť na obrázku 3.1, systém BeeeOn je zložený z niekoľkých úrovní. Na najnižšej úrovni sa nachádzajú samotné hardvérové prvky zodpovedné za monitorovanie (*senzory*) alebo kontrolu (*aktory*) fyzického prostredia. Tieto zariadenia priamo komunikujú s bránou typicky prostredníctvom siete typu PAN (*Personal Area Network*). Jedná sa väčšinou o bezdrôtovú rádiovú komunikáciu, ale nie je to pravidlom.

Brána (*gateway*) je hlavným prístupovým bodom v domácnosti, ako už bolo spomenuté, jej úlohou je komunikácia s najnižšími prvkami systému, pričom celú komunikáciu ďalej propaguje na server. K serveru je brána pripojená prostredníctvom siete Internet. Tento prvok systému je teda možné považovať za akési rozhranie medzi senzorickou sieťou a serverom.

Na najvyššej úrovni sa nachádza používateľ, ktorý formou používateľskej aplikácie má možnosť interagovať so zvyškom celého systému.



Obr. 3.1: Architektúra systému BeeeOn

3.2 Server

Server tvorí jadro celého systému a nesie zodpovednosť za viaceré služby. Komunikuje a riadi brány pripojené do systému, zabezpečuje perzistentné ukladanie dát a poskytuje podporu pre používateľské aplikácie. Server je určený pre beh na stroji s architektúrou *x86* a operačným systémom GNU/Linux.

Z hľadiska systému má server dve hlavné úlohy. Prvou úlohou je komunikácia a obsluha všetkých pripojených brán. Spracuváva požiadavky prichádzajúce z brány a v prípade potreby, dokáže delegovať príkazy pre bránu. Druhá úloha serveru je komunikácia s používateľskou aplikáciou (napr. Grafická aplikácia pre smartphone) opäť prostredníctvom siete Internet.

3.2.1 Architektúra

Architektonicky sa jedná o viacvrstvú architektúru. Takáto architektúra má výhody ako určenie jednoznačnej zodpovednosti každej vrstvy alebo výmena implementácie nejakej vrstvy za inú bez výrazného ovplyvnenia ostatných vrstiev. V systéme rozlišujeme 3 vrstvy:

- Aplikačná vrstva - Hlavnou úlohou je zabezpečenie komunikácie s ostatnými časťami systému, ako sú brány a používateľské aplikácie.
- Servisná vrstva - Táto vrstva, nazývaná tiež *business* vrstva, implementuje internú logiku aplikácie.
- Dátová vrstva - Slúži na oddelenie logiky aplikácie od konkrétnej použitej databázy.

3.2.2 GW Server

Z pohľadu zodpovednosti vieme ešte Server rozdeliť na 2 samostatné komponenty. Prvou z nich je GW Server (*Gateway Server*). Jej zodpovednosť je komunikácia s bránami. Spôsob

komunikácie medzi bránou a serverom je popísaný v bakalárskej práci *Server pro sběr senzorických dat a řízení aktivních prvků* od Jozefa Halaja z roku 2017 [11]. Na komunikačnej úrovni je použitý protokol WebSocket, ktorý vytvára obojstranné spojenie medzi bránou a serverom. Šifrovanie spojenia je zabezpečené vrstvou TLS, pri čom sa pri spojení overuje serverový aj klientský certifikát. Na aplikačnej úrovni sa používa protokol založený na báze JSON [1] navrhnutý pre toto konkrétne použitie. Spojenie iniciuje vždy brána, ktorá po nadviazaní spojenia vytvorí registračnú správu so základnými údajmi. Ďalšia komunikácia je možná až po potvrdení registračnej správy serverom.

3.2.3 UI Server

Druhou komponentou Serveru je takzvaný UI Server (*User Interface Server*), jeho zodpovednosťou je komunikácia s používateľskou aplikáciou. V súčasnosti sú podporované 2 spôsoby komunikácie používateľskej aplikácie a BeeeOn serveru. V prvom prípade ide o protokol na báze jazyku XML, ktorý je podporovaný prevažne z kompatibilných dôvodov, pretože je tento spôsob komunikácie implementovaný už len udržiavanou aplikáciou pre zariadenia s operačným systémom Android.

Druhý a hlavný podporovaný spôsob komunikácie je založený na technológii REST. REST s použitím HTTP dotazov umožňuje pristupovať a modifikovať vzdialené zdroje [9].

3.3 Brána

Brána je dôležitý prvok systému, pretože je zodpovedná za integráciu systémov tretích strán do systému BeeeOn. Jedná sa o vstupný bod do inteligentnej domácnosti.

Zo softvérového hľadiska sa o všetky úkony stará aplikácia bežiaca v používateľskom priestore (*user space*). Pri vývoji je kladený dôraz na prenositeľnosť a multiplatformnosť aplikácie. Aplikácia je navrhnutá tak, aby mohla podporovať viaceré skupiny zariadení formou modulov tak, ako je to znázornené na obrázku 3.2. Hlavnou podstatou brány sú moduly, pričom každý modul spravuje nejakú špecifickú skupinu zariadení. Skupinou zariadení môžeme chápať zariadenia, ktoré zdieľajú konkrétnu použitú technológiu (spôsob fyzickej komunikácie, komunikačný protokol, apod.). Môže sa jednať napríklad o zariadenia komunikujúce prostredníctvom Bluetooth, Z-Wave, ale aj ďalších. Každé podporované zariadenie v systéme musí patriť do určitej skupiny. V súčasnom systéme sú tieto moduly súčasťou hlavnej aplikácie, takže ich pridanie alebo odobranie má kompilačnú závislosť. Do budúcnosti je však naplánované rozdeliť tieto časti aplikácie na samostatné služby.

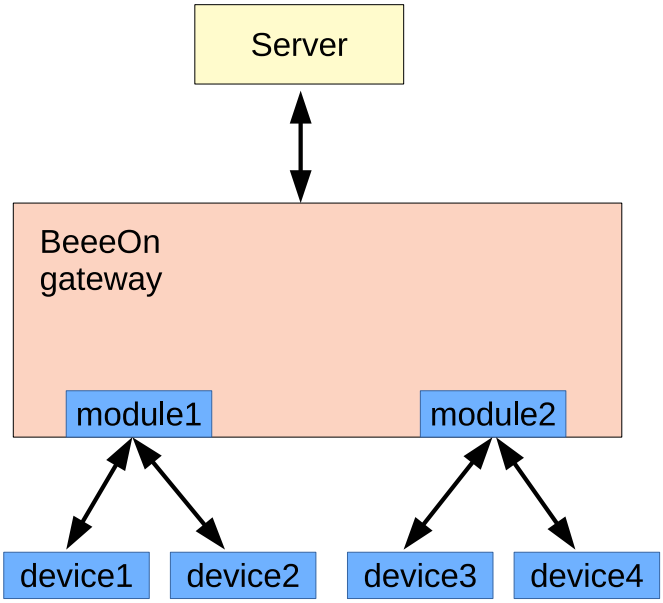
Každej bráne v systéme patrí unikátny 64bitový identifikátor *Gateway ID*. Tento identifikátor je obsiahnutý v *common name* klientského TLS certifikátu, ktorý je unikátny pre kažú bránu.

Každé zariadenie v systéme BeeeOn je identifikované unikátnym globálnym identifikátorom nazvaným *Device ID*, ktoré má šírku 64 bitov. Prvých 16 bitov tohoto identifikátoru tvorí takzvaný *Device Prefix*, ktorý špecifikuje, do ktorej skupiny zariadení dané zariadenie patrí.

3.3.1 Hardvér

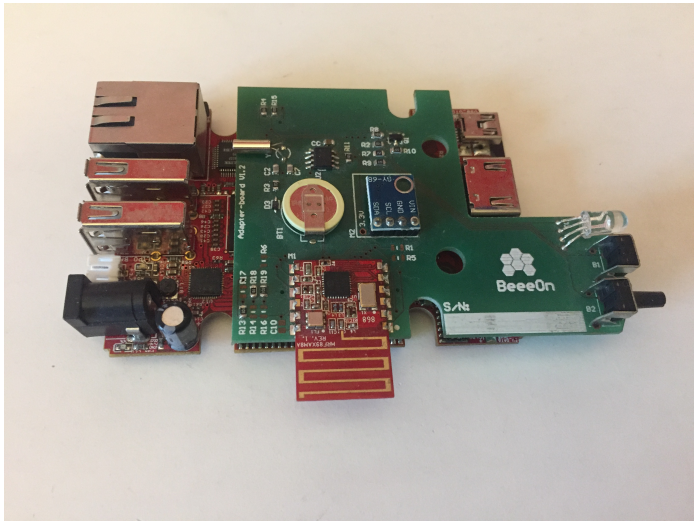
Aplikácia brány je primárne vyvýjaná pre beh na špeciálnom dedikovanom vstavanom zariadení o rozmeroch približne 11cm × 6cm zobrazenom na obrázku 3.3. Jedná sa o minipo-

[1] https://www.json.org/



Obr. 3.2: Modulárny koncept aplikácie brány

čítač A10-OLinuXino-LIME s procesorom Cortex-A8 postavenom na architektúre ARMv7. Na zariadení beží vlastný, pre účely projektu upravený operačný systém s jadrom Linux. Multiplatformnosť aplikácie okrem toho umožnuje beh aj na stroji s inou mikroprocesorovou architektúrou (napr. x86). Hlavnou požiadavkou pre beh je prostredie s operačným systémom GNU/Linux.

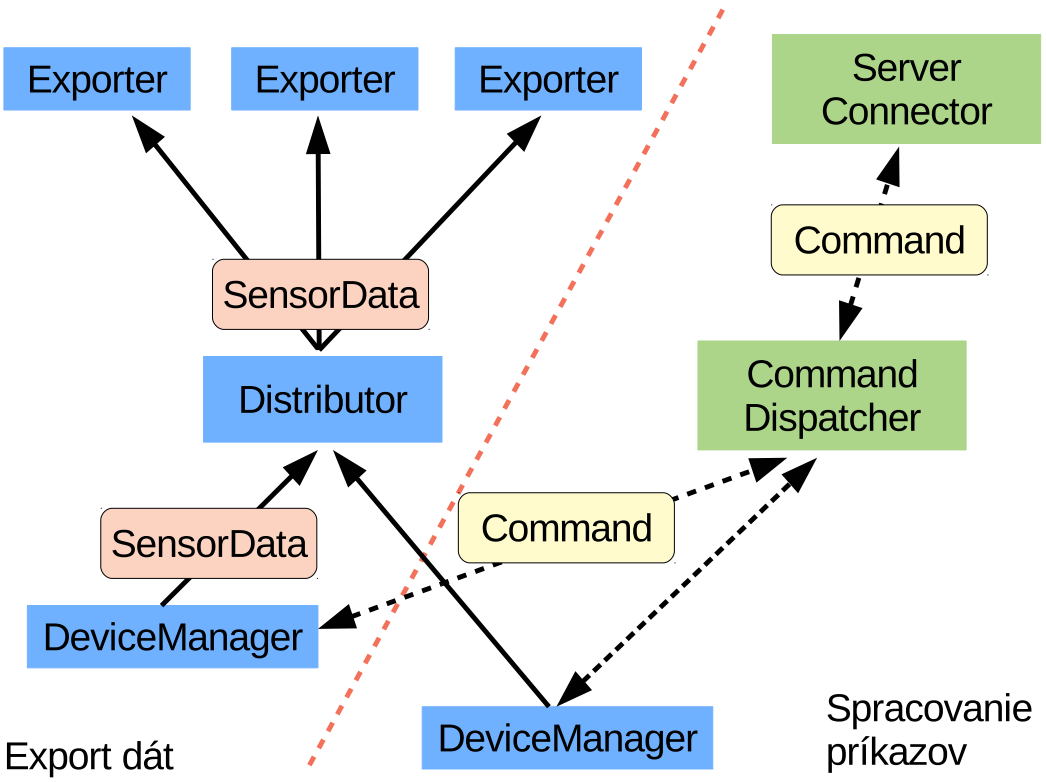


Obr. 3.3: Hardvér BeeeOn brány

3.3.2 Architektúra aplikácie

Z pohľadu najvyššej úrovne je architektúra aplikácie zodpovedná za export nazbieraných dát a riadenie prvkov formou interného systému príkazov tak, ako je to možné vidieť na obrázku 3.4. Definujme si tu základné rozhrania systému:

- `DeviceManager` – Nejde o čisté rozhranie, ale o abstraktnú triedu, ktorej zodpovednosť je riadiť nejakú špecifickú skupinu zariadení. Triedy, ktoré špecifickým spôsobom implementujú túto abstraktnú triedu môžeme označovať ako správcov zariadení.
- `Exporter` – Zodpovednosť tohoto rozhrania je exportovanie senzorických dát implementačne špecifickým spôsobom.
- `Distributor` – Trieda implementujúca toto rozhranie nesie zodpovednosť za distribúciu senzorických dát, dokáže registrovať objekty implementujúce rozhranie *Exporter*
- `CommandDispatcher` – Táto trieda má zodpovednosť za riadenie príkazov systému.

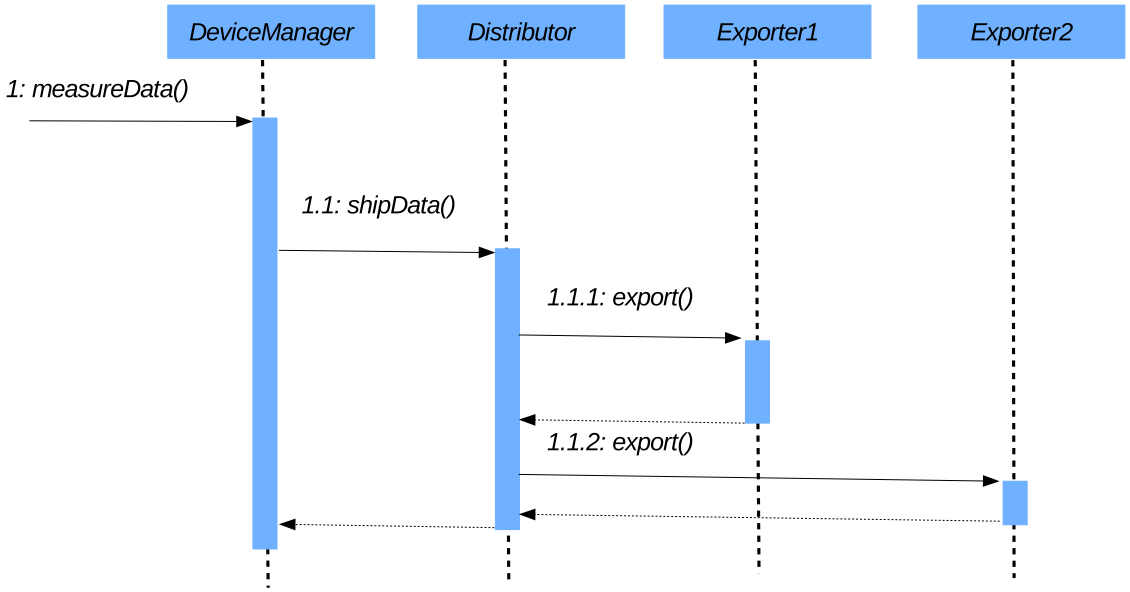


Obr. 3.4: Top-level architekúra hlavnej aplikácie brány

3.3.3 Proces exportu dát

Dôležitou úlohou brány je export nazbieraných senzorových dát. Do tohoto procesu sa zapája niekoľko komponent. Tento proces je znázornený na diagrame 3.5. Celý proces začína zmeraním nejakej senzorickej hodnoty na fyzickom zariadení. Správca zariadení, ktorý

dané zariadenie spravuje túto správu predá objektu s rozhraním *Distributor*, ktorý sa v systéme typicky nachádza jeden. *Distributor*, ktorý má zaregistrované objekty rozhrania *Exporter* týmto objektom správu dalej predá. Každý objekt rozhrania *Exporter* následne správu exportuje predom špecifikovaným spôsobom a formátom. Môže sa jednať napríklad o export pomocou protokolu MQTT, export na BeeeOn server alebo ďalšie podporované formáty.



Obr. 3.5: Architektúra exportu dát

3.3.4 Proces spracovania príkazov

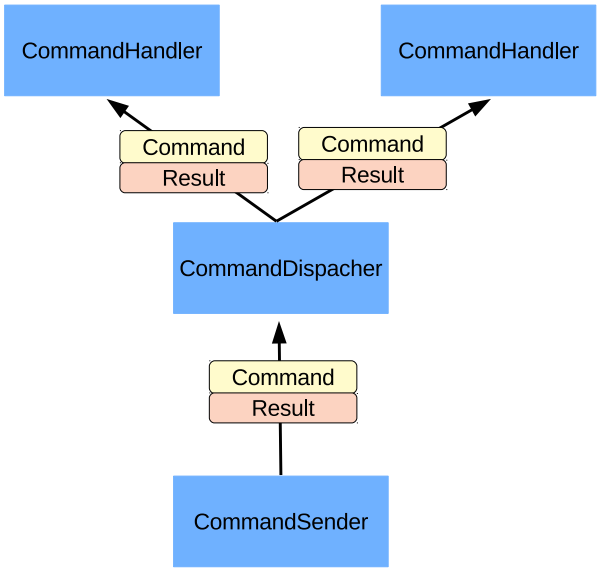
V aplikácii brány existuje interný systém riadenia príkazov. Tento systém zabezpečuje vysokoúrovňovú asynchrónnu komunikáciu medzi prvkami systému. Do týchto príkazov sa typicky transformujú rôzne požiadavky zo servera, ako je napríklad požiadavka prepnutia brány do párovacieho režimu, prijatie zariadenia ako napárovaného, alebo nastavenie nejakého aktora.

Vystupujú tu tri hlavné komponenty, ktoré sú znázornené aj na obrázku 3.6:

- `CommandSender` – Objekty implementujúce toto rozhranie dokážu posielať príkazy do systému prostredníctvom objektu triedy `CommandDispatcher`.
- `CommandHandler` – Objekt, ktorý implementuje toto rozhranie je schopný prijímať správy. Pre každý príkaz sa najskôr overí výsledok predikátu *accept()*, ktorý určuje, či objekt dokáže prijať daný typ príkazu. V kladnom prípade dôjde k spracovaniu príkazu pomocou metódy *handle()*, vrámci ktorej je povinnosť nastaviť príkazu výsledok a pridať odpoveď s určitým stavom (úspech, neúspech).
- `CommandDispatcher` – Inštancia tejto triedy sa nachádza v centre procesu. Je schopný registrácie objektov rozhrania `CommandHandler`, ktorým preposiela príkazy metódou *dispatch()*, ktorú evokuje objekt implementujúci abstraktnú triedu `CommandSender`.

Ukážku tohoto procesu znázorňuje diagram 3.7. Proces začína spracovaním nejakej požiadavky (napríklad zo serveru). Objekt triedy `CommandSender` následne zašle príkaz in-

štancii triedy `CommandDispatcher`, ktorá je zodpovedná za rozoslanie príkazu. Spracovanie príkazu prebieha v dvoch krokoch. `CommandDispatcher` musí najskôr zistiť, kto všetko prijatému príkazu rozumie prostredníctvom metódy *accept()*. Registrované objekty typu `CommandHandler`, ktoré príkaz prijímu ho následne dostanú na spracovanie prostredníctvom metódy *handle()*.



Obr. 3.6: Architektúra príkazového systému

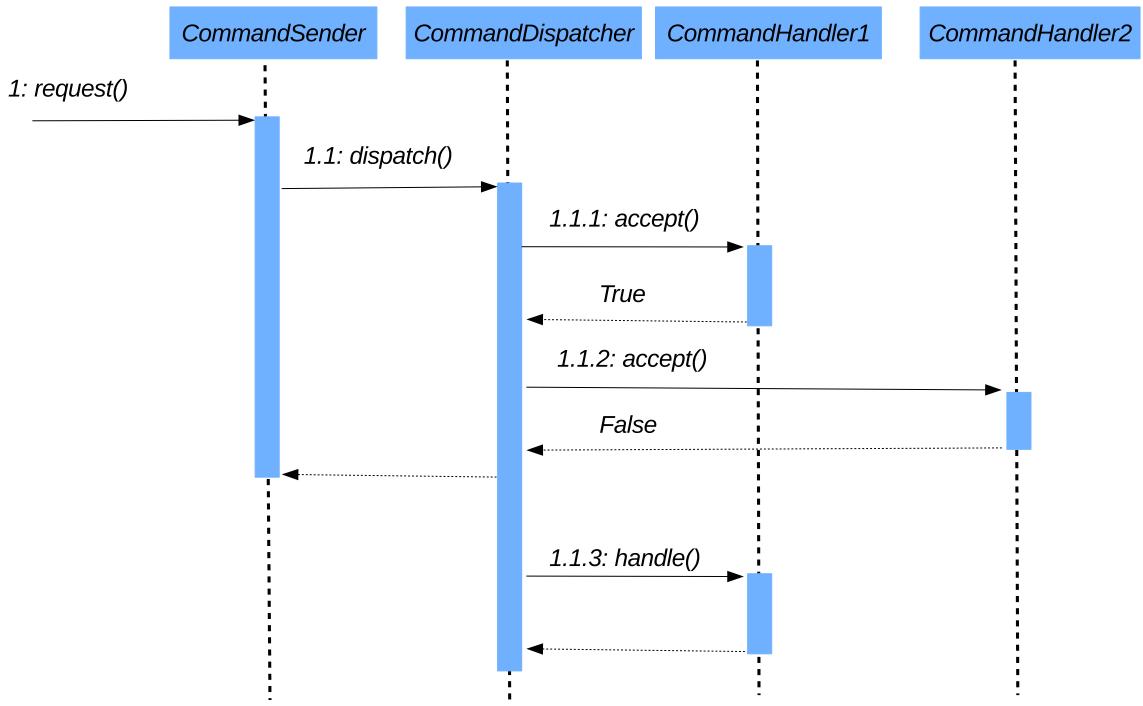
Objekty, ktoré implementujú abstraktnú triedu `CommandSender` typicky implementujú aj rozhranie `CommandHandler`, aby boli schopné príkazy prijímať aj posielať. Systém je navrhnutý s ohľadom na to, že príkazy môžu trvať dlhšiu dobu, počas ktorej sa stav spracovania príkazu môže meniť. Príkladom môže byť príkaz pre zaradenie na otváranie dverí. Dvere sa nikdy neotvoria v diskrétnom okamžiku, stavu otvorenia predchádza stav otvárania, ktorý môže trvať dlhšiu dobu, alebo môže uplne zlyhať. Aby systém dokázal používateľa patrične informovať, musí dokázať tieto stavy rozlíšiť.

3.3.5 Testovacie centrum

Testovacie centrum je komponenta systému brány, ktorá dokáže vytvárať a distribuovať príkazy do systému prostredníctvom jednoduchej textovej konzoly. Okrem toho dokáže emulovať pripojenie aplikácie na server tým, že implementuje spracovanie príkazov určených pre server. Používa sa výhradne pre testovacie a ladiace účely.

3.3.6 Virtuálne zariadenia

Systém brány obsahuje špeciálnu triedu implementujúcu abstraktnú triedu `DeviceManager`, ktorá dokáže emulovať reálne zariadenia predom definovaného a konfigurovateľného typu. Senzorické dáta v tomto prípade nie sú merané ale generované. Spôsob generovania je tak isto konfigurovateľný, pričom je hodnoty možné generovať konštantne, náhodne zo zvoleného rozsahu alebo s použitím periodickej funkcie sínus. Virtuálne zariadenia slúžia predovšetkým na testovacie a demonstračné účely. Takéto zariadenia je možné nadefinovať a konfiguro-



Obr. 3.7: Ukážka procesu spracovania príkazov.

vať vo formáte INI. Konfigurácia jedného virtuálneho zariadenia môže vyzerať tak, ako je znázornené na 3.1

```
[virtual-device1]
enable = yes
paired = yes
refresh = 60
device_id = 0xa300000000000002
vendor = VUT / RehiveTech
product = Air pressure sensor
module0.type = pressure
module0.min = 0
module0.max = 15
module0.generator = random
module0.reaction = none
```

Výpis 3.1: Ukážková konfigurácia virtuálneho zariadenia.

Z konfigurácie 3.1 vidíme, že sa jedná o virtuálne zariadenie na meranie atmosferického tlaku, ktoré je dopredu napárované, takže ihneď po spustení začne generovať každých 60 sekúnd jednu náhodnú hodnotu z rozsahu 0 až 15. Pre zvyšok systému je toto transparentné a nedokáže rozoznať, či ide o reálnu veličinu alebo generovanú hodnotu. Z hľadiska testovania sa preto jedná o potencionálne veľmi užitočnú komponentu.

3.3.7 Emulátory zariadení

Hoci sa nejedná priamo o súčasť aplikácie ani konfigurácie, v projekte brány v súčasnosti existujú emulátory niektorých reálnych zariadení, ktoré svojou činnosťou emulujú prítomnosť a dostupnosť týchto zariadení. Na rozdiel od virtuálnych zariadení teda nie sú priamo súčasťou brány a sú nerozlíšiteľné oproti skutočným zariadeniam. Z pohľadu testovania majú veľký význam, pretože pomocou nich môžeme testovať samotnú špecifickú implementáciu abstraktnej triedy `DeviceManager`, ktorá ponúka interakciu s nejakou špecifickou skupinou zariadení.

V súčasnosti existujú v projekte tieto emulátory z rodiny Belkin WeMo[2]:

- *Belkin WeMo Switch* – Vzdialene ovládateľná elektrická žástrčka. Umožňuje vypnutie, zapnutie a rovnako aj reportuje svoj stav.
- *Belkin WeMo Link* – Toto zariadenie umožňuje ovládať inteligentné LED žiarovky značky Belkin WeMo. Pre asociovanú žiarovku umožňuje zapnutie, vypnutie a zmenu intenzity svietenia.
- *Belkin WeMo Dimmer* – Vzdialene ovládateľný regulátor osvetlenia. Umožňuje vypnutie a zapnutie svetiel, ako aj nastavenie intenzity svietenia.

Pri tvorbe takýchto emulátorov sa môžeme všeobecne stretnúť s množinou problémov, ako môže byť napríklad uzavrenosť knižníc, ktoré tieto technológie podporujú, problematické emulovanie niektorých potrebných vstupno/výstupných zariadení. Niektoré technológie môžu byť naviac komplexne zložitejšie, takže by musel byť verifikovaný aj samotný emulátor pre vierohodné výsledky pri jeho použití.

3.4 Vývoj systému

Na to aby bolo možné testovaciu sadu navrhnúť a integrovať do vývoja efektívne, je potrebné poznať spôsob akým sa projekt vyvýja, čomu sa bude venovať táto podkapitola.

Vývoj systému v súčasnosti prebieha prevažne inkrementálnym spôsobom. Pre správu zdrojových kódov sa používa rozšírený verzovací systém *Git* [3]. Komunikácia ohľadom vývoja prebieha predovšetkým prostredníctvom mailing listov, do ktorých sú prihásení vývojári a vedúci projektu.

3.4.1 Revízia kódu

V projekte BeeeOn je zavedený striktný proces revízie kódu. Každý kus kódu, ktorý sa dostane do hlavnej vetvy projektu musí najskôr prejsť týmto procesom, preto je potrebné sa s týmto procesom zoznámiť a definovať ho.

Revízia kódu je úzko spojená s použitým verzovacím systémom *Git*. Najmenšou jednotkou samostatnej revízie je vždy jeden *commit* verzovacieho systému. Na revíziu sa vždy typicky posiela väčší logický celok nazvaný *patch-set*. Ide o chronologicky zoradenú postupnosť commitov, ktoré spolu typicky logicky súvisia, napríklad pridávajú systému novú funkcionalitu, opravujú nejaký typ alebo celú skupinu problémov, či refaktorujú kód. Platí, že žiadny commit by nemal negatívne ovplyvniť preložiteľnosť ani funkcionalitu systému, ani obsahovať žiadne vedľajšie efekty, ktoré nie sú definované v jeho popise.

[2] http://www.belkin.com/us/Products/home-automation/c/wemo-home-automation/
[3] https://git-scm.com/

Každý vývojár má isté, predom definované, zodpovednosti a úlohy. Celý proces začína tým, že autor kódu žiada o jeho revíziu odoslaním *patch-setu* do vývojového mailing listu. K tomu typicky prikladá vodiaci email (*cover letter*), v ktorom vlastnými slovami popisuje čoho sa daný *patch-set* týka. Revíziu môže dobrovoľne vykonať ktorýkoľvek člen tímu, typicky je však povinný kód revidovať správca daného repozitáru, ktorého sa kód týka. Správca nesie najvyššiu zodpovednosť za to, čo sa začlení do hlavnej vetvy projektu. Výsledkom takejto revízie môžu byť poznámky a vecné pripomienky, návrhy na zlepšenie, a podobne.

V krajnom prípade môže byť celý kód odmietnutý, ak už nedáva zmysel. Autor kódu by sa mal (minimálne zo slušnosti) ku každej pripomienke a poznámke vyjadriť, že ju buď zapracoval alebo s ňou nesúhlasí s dôvodnou vecnou argumentáciou. V tomto ohľade môže vzniknúť diskusia, do ktorej sa má možnosť opäť zapojiť ktokoľvek z vývojového tímu. U zložitejších problémov je dokonca takáto diskusia žiadúca. Diskusia by mala byt vedená v duchu vecnej argumentácie a vzájomného rešpektu, subjektívnym a osobným názorom je potrebné sa vyvarovať. Proces má iteratívny charakter a po spracovaní všetkých poznámok, respektíve po vyjasnení a uzavretí diskusie autor pošle ďalšiu verziu daného kódu. Počas tohoto procesu sa do samostaných commitov pridávajú značky, pod ktoré sa podpisuje osoba, na ktorej podnet sa značka vytvorila. Značky majú následujúci formát: “Značka: Meno Priezvisko <email>”, teda napríklad “Signed-off-by: John Doe <john.doe@example.org>”. Sémantika značiek je nasledujúca:

- Signed-off-by: Jedná sa o podpis autorov daného kódu. Pridáva sa pri vytvorení a prípadne aj modifikácii daného commitu.
- Reviewed-by: Túto značku pridáva ktokoľvek, kto revidoval daný kód, nenašiel v ňom žiadne zjavné chyby a nemá k nemu už žiadne ďalšie výhrady
- Acked-by: Pridáva zodpovedná osoba (typicky správca repozitáru). Značí, že kód je dostatočne kvalitný pre začlenenie do hlavnej vetvy repozitára.
- Tested-by: Pridáva ktokoľvek, kto daný kód aplikoval, prekladal a otestoval bez objavených chýb. Potvrdzuje, že daná funkcionalita funguje alebo daná chyba, ktorú commit opravuje je vyriešená.
- Reported-by: Slúži na pridanie kreditu osobe, ktorá nahlásila nejakú softvérovú chybu. Môže byť pridaná do commitu, ktorý opravuje túto chybu.
- Suggested-by: Slúži na pridanie kreditu osobe, ktorá vytvorila nejaký podnet, na základe ktorého vznikol daný commit.

Po pridaní značky *Acked-by* je možné commit začleniť do hlavnej vetvy repozitáru. Typicky sa začlenuje celý *patch-set*, v ktorom každý jednotlivý commit dostal túto značku. Začlenením celý proces revízie kódu končí. Kým sa tak stane, typicky sa vykoná 5 až 10 iterácií.

Značky sa pridávajú na spodok popisu daného commitu v chronologickom poradí tak, ako boli pridané od prvej značky na začiatku a poslednej značky na konci. Typicky teda postupnost značiek začleneného commitu začína značkou *Signed-off-by* a končí značkou *Acked-by*.

3.5 Implementačné detaily

Projekt je implementovaný prevažne v jazyku C++ s podporou štandardu *C++11* a využitím všeobecnej knižnice *Poco*. V súčasnosti je zdrojový kód projektu rozdelený do 3 hlavných repozitárov verzovacieho systému:

- Gateway – V tomto repozitári sa nachádzajú zdrojové kódy aplikácie brány a konfiguračné súbory.
- Server – Podobne ako v prípade brány aj serverová časť má vyhradený vlastný repozitár pre zdrovojé kódy a konfiguračné súbory serveru.
- Base – Tento repozitár slúži hlavne pre generický kód, ktorý je zdieľaný ako submodul verzovacieho systému. Môže byť teda použitý kdekoľvek naprieč celým systémom.

3.6 Prípady použitia systému BeeeOn

Táto kapitola bude popisovať a definovať štandardné prípady použitia systému, ktoré bude potrebné pokryť testovacou sadou.

3.6.1 Registrácia brány

V prípade prvého pripojenia novovytvorenej brány, ktorá sa ešte nenachádza v systéme musí server po úspešnom spracovaní registračnej správy aplikačného protokolu pridať túto bránu aj s ďalšími údajmi do perzistentnej databázy. Aplikácia brány po nadviazaní komunikácie na server automaticky posiela správu typu `gateway_register`, ktorá obsahuje identifikačné číslo brány, IP adresu použitého sieťového rozhrania a verziu aplikácie. V prípade, že je všetko v poriadku, server potvrdí túto správu pomocou správy typu `gateway_accepted`. V opačnom prípade je server nútený uzavrieť spojenie.

3.6.2 Spravovanie brán používateľom

Požívateľ môže manipulovať s viacerými bránami systému s rôznymi oprávneniami. Manipuláciu môžeme rozdeliť do týchto prípadov:

- Registrácia vlastníka – Aby bolo možné bránu používať, musí mať vlastníka. Jedná sa o používateľa, ktorému brána patrí a má najvyššie právomoci. Vlastníkom je automaticky používateľ, ktorý si bránu pridá ako prvý. V typickej situácii registrácia vlastníka nasleduje prvú registráciu brány v systéme. Používateľ teda zapojí bránu a cez používateľskú aplikáciu si ju zaregistruje.
- Odstránenie brány – Vlastník brány môže bránu zo systému odstrániť, čím sa odstránia všetky dáta o jej existencii, napárované zariadenia a história senzorických dát.
- Správa používeľov – Brána môže byť zdieľaná medzi viacerými používateľmi, správca brány a vlastník môžu bráne priradiť používateľov s rôznymi právami.

3.6.3 Párovanie

Proces pridávania nových zariadení začína aktiváciou párovacieho režimu. Párovací režim zvyčajne trvá približne 2 minúty. Počas tejto doby sa každá inštancia triedy implementujúcej `DeviceManager` na bráne snaží vyhľadávať nové zariadenia. V prípade, že správca

objaví nové zariadenie, pošle správu o detailoch tohoto zariadenia serveru. Takáto správa obsahuje identifikačné číslo daného zariadenia, výrobcu, názov produktu a zoznam podporovaných modulov a ich atribútov. Server každé prijaté podporované zariadenie propaguje naspäť používateľovi, ktorý má k dispozícii zoznam novoobjavených zariadení s ich detailom. Následne má možnosť si vybrať zariadenie, ktoré chce na bránu napárovať, čím dôjde k finálnemu priradeniu nového zariadenia pre bránu. Od tohoto momentu zariadenie exportuje senzorové dáta, respektíve, ak sa jedná o aktívny prvok, je možné meniť jeho stav.

Proces odpárovania je jednoduchší, používateľ označí, ktoré zariadenie chce odpárovať, čo sa spropaguje na server a zmažú sa údaje o tomto zariadení.

3.6.4 Zbieranie a export dát

Brána počas svojho behu zbiera senzorové dáta z napárovaných zariadení a exportuje ich na server, ktorý ich ďalej ukladá na perzistentné úložisko. Pri tomto procese existuje riziko napríklad straty spojenia na server, v horšom prípade dokonca strata napájania hardvéru brány. Aplikácia brány v sebe obsahuje mechanizmy, ktoré sa toto snažia riešiť vo forme vlastnej vyrovnávacej pamäte, kde sa ukladajú dáta počas nemožnosti doručenia serveru. Po znovuobnovení konektivity sú dáta aj postupne vyexportované na server. Pretože dáta majú svoje vlastné časové značky, vieme presne určiť, že sa jedná o historické alebo aktuálne hodnoty.

3.6.5 Riadenie aktorových prvkov

Používateľ môže vyžiadať zmenu stavu nejakého aktívneho prvku. Môže ísť napríklad o rozsvietenie svetiel, otvorenie okien, zvýšenie vykurovacej teploty a podobne. O stave tejto operácie a prípadnom úspechu či neúspechu by mal byť používateľ, pokiaľ je to možné informovaný. V praxi to bohužial, nie je vždy možné overiť, pretože niektoré aktorové zariadenia len pasívne počúvajú a zmenu svojho stavu nereportujú naspäť.

3.7 Testovanie v projekte

V súčasnom systéme existujú testy na jednotkovej úrovni s využitím knižnice *CppUnit*. Jednotkové testy prechádzajú revíziou spoločne so samotným kódom. V projekte BeeeOn server existujú aj integračné funkčné testy, písané v jazyku Python, s využitím knižnice Python.unittest.

Testovanie integrácie a systému prebieha prevažne manuálne po uzavretí majorinyných verzií. Tento proces je pomerne pracný a časovo náročný.

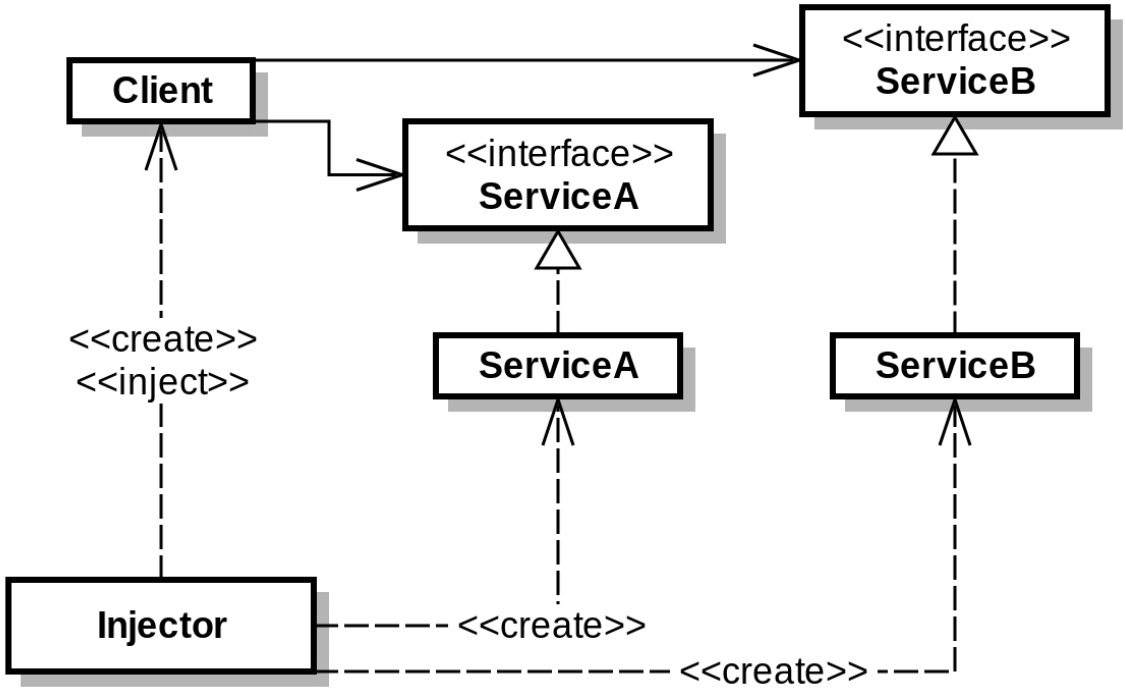
3.8 BeeeOn aplikácie

Aplikácie BeeeOn sú písané prevažne v jazyku C++. Pre kompiláciu aplikácií sa využíva otvorený prekladový systém CMake. Projekt je špecifický tým, že využíva techniku v objektovo orientovanom programovaní zvanú vkladanie závislostí (*Dependency injection*), ktorá je podmnožina konceptu *Inversion of Control* [4]. Táto technika aj s využitím v projekte BeeeOn bude vysvetlená v tejto podkapitole.

3.8.1 Vkladanie závislostí

Vkladanie závislostí (*Dependency injection*) je technika v objektovo orientovanom programovaní, pri ktorej je hlavným princípom odobranie zodpovednosti tried za nastavenie závislostných objektov, ktoré sú potrebné k ich činnosti. Táto zodpovednosť je presunutá typicky na nejakú externú utilitu, ktorá sa označuje ako *Injector*. *Injector* je teda zodpovedný za dodanie týchto závislostí objektom a typicky aj za ich vytvorenie predom špecifikovaným spôsobom. Princíp fungovania je možné vidieť na obrázku 3.8. V tejto ukážke máme nejakú triedu `Client`, ktorá pre svoju činnosť potrebuje inštanciu triedy implementujúcu rozhranie `ServieceA` a inštanciu triedy, ktorá implementuje rozhranie `ServiceB`. Injector teda vytvorí samotnú inštanciu triedy `Client`, inštanciu triedy `ServiceA` a inštanciu triedy `ServiceB`. Po vytvorení Injector dodá inštancii triedy `Client` potrebné vytvorené závislosti.

Výhody použitia sú napríklad lepšie udržovateľný a rozšíriteľný kód či jednoduchšia tvorba jednotkových testov. K nevýhodám patrí napríklad pomerne veľká kódová réžia v prípade menších aplikácií [14].

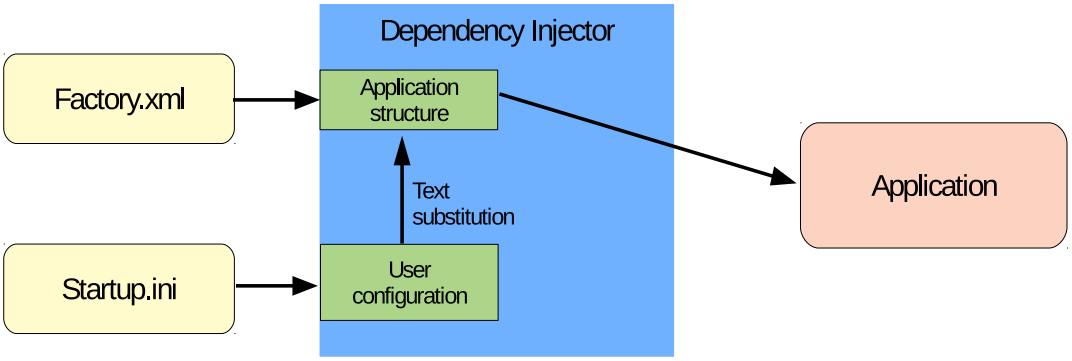


Obr. 3.8: Princíp fungovania techniky vkladania závislostí[3].

3.8.2 Vkladanie závislostí v BeeeOn

Projekt BeeeOn techniku vkladania závislostí implementuje vlastným spôsobom v jazyku C++ bez použitia externých knižníc určených špeciálne pre toto použitie. Štruktúra samotnej aplikácie, objektov a ich závislostí nie je známa v dobe kompilácie a vytvára sa až v dobe samotného behu. Štruktúra aplikácie je definovaná pomocou textových konfiguračných súborov vo formáte *XML*, ktoré sú označované ako *factory.xml*. Proces vytvorenia aplikácie je znázornený na obrázku 3.9, kde môžeme vidieť, že konfigurácia je rozdelená na používateľskú konfiguráciu vo forme súborov formátu INI, tá slúži hlavne pre konfigurovanie položiek vo *factory.xml*. Svojím spôsobom sú teda zamerané viac používateľsky a

slúžia pre konfiguráciu samotných komponentov, ako napríklad použitý port pre server. XML konfigurácia oproti tomu definuje štruktúru aplikácie a nie sú primárne určené pre konfiguráciu používateľom. Výhoda tohoto riešenia spočíva vo flexibilite, pretože je možná kompletná zmena celej aplikácie bez nutnosti akejkoľvek rekompilácie. Na druhej strane prichádzame kontrolu validity konfigurácie v dobe prekladu. Tento nedostatok je čiastočne riešený skriptom, ktorý generuje grafickú reprezentáciu konfigurácie a v prípade napríklad syntaktických chýb vieme automatizovane odhaliť problém.



Obr. 3.9: Konfigurácia *Injectoru* pre vytovrenie aplikácie.

Program dostane v argumentoch cestu ku konfiguračným súborom, pomocou ktorých Injector vytvorí aplikáciu. Definícia objektu je znázornená v ukážke 3.2.

```
<instance name="virtualDeviceManager" class="BeeeOn::VirtualDeviceManager">
        <set name="distributor" ref="distributor"/>
        <set name="commandDispatcher" ref="commandDispatcher"/>
        <set name="file" text="${vdev.ini}"/>
</instance>
```

Výpis 3.2: Ukážka definície objektu pre vkladanie závislostí.

Ako môžeme vidieť v ukážke 3.2, každý objekt má svoj identifikátor (*name*), ktorý je možné použiť pre jeho odkazovanie a označenie triedy, ktorá má byť použitá pre vytvorenie tohoto objektu. Okrem toho má špecifikované atribúty (teda závislosti), ktoré musia byť uspokojené pre jeho fungovanie. Môže sa jednať o textové reťazce, čísla s pevnou aj pohyblivou rádovou čiarkou, časové hodnoty, zoznamy a v neposlednmom rade aj odkazy iných objektov. Tieto hodnoty môžu byť aj parametrizovateľné, ako môžeme v ukážke 3.2 vidieť položku *file*, ktorá je textový reťazec, nachádzajúci sa v používateľskom konfiguračnom súbore. *Injector* pred svojím behom vykoná textovú substitúciu. Hlavná funkcia celej aplikácie je vo výsledku, ako je znázornené v ukážke 3.3, vyzerá jednoducho.

```
int main(int argc, char **argv)
{
        About about;
        DIDaemon::up(argc, argv, about);
}
```

Výpis 3.3: Hlavná funkcia BeeeOn aplikácie.

V ukážke 3.3 vidíme ešte využitie štruktúry `About`. Táto štruktúra v sebe zahŕňa základné údaje o aplikácii ako je verzia, popis aplikácie a požiadavky na verziu použitej knižnice *Poco*.

3.8.3 Argumenty príkazového riadku

V tejto podsekcii popíšem rôzne prepínače a parametre príkazového riadku, ktoré je možné použiť pre spustenie BeeeOn aplikácie.

- -c <cesta> – Povinný parameter pre špecifikáciu cesty k používateľskému konfiguračnému súboru formátu INI.
- -D<kľúč=hodnota> – Voliteľný parameter pomocou ktorého je možné vynútiť nejaký parameter konfiguračného súboru na špecifikovanú hodnotu. Napríklad pre spustenie aplikácie brány s portom pre pripojenie na server 1010 by vyzeralo jeho spustenie takto: `beeeon-gateway -c startup.init -Dgws.port=1010`.
- -N<číslo procesu> – Voliteľný parameter ktorým je možné špecifikovať čislo procesu, ktorému bude oznámená úspešná inicializácia aplikácie. Oznamovanie prebieha zaslaním signálu typu `SIGTERM` špecifikovanému procesu.

Kapitola 4

Návrh automatických testov

Táto kapitola sa bude venovať návrhu automatických testov a definuje spôsob ich integrácie a použitia vo vývoji. Nakoľko je tvorba testov na jednotkovej úrovni v projekte už zavedená, bude v tejto kapitole vynechaná.

4.1 Integračné testy

V prípade integračných testov je potrebné rozhodnúť aké veľké časti chceme integrovať. Integráciu som sa rozhodol rozdeliť na časti, ktoré tvoria samostatne bežiaci proces, pri týchto testoch sa teda bude vždy používať kompletný binárny súbor určený pre nasadenie v spojení so základnou konfiguráciou, ktorú budeme len modifikovať prostredníctvom používateľskej konfigurácie.

4.1.1 Testovanie brány

Pre to, aby bolo možné testovať integráciu, potrebujeme vedieť akým spôsobom môžeme s integrovaným celkom interagovať. Možnosti komunikácie testovacieho prostredia s aplikáciou brány sú znázornené na obrázku 4.1.

Na spodnej časti sa nachádzajú zariadenia, predovšetkým sú pre nás dôležité virtuálne zariadenia, ktoré môžeme špecifikovať pomocou konfigurácie. Okrem toho sa tu môžu nachádzať potencionálne aj emulátory skutočných zariadení, ak takéto emulátory existujú.

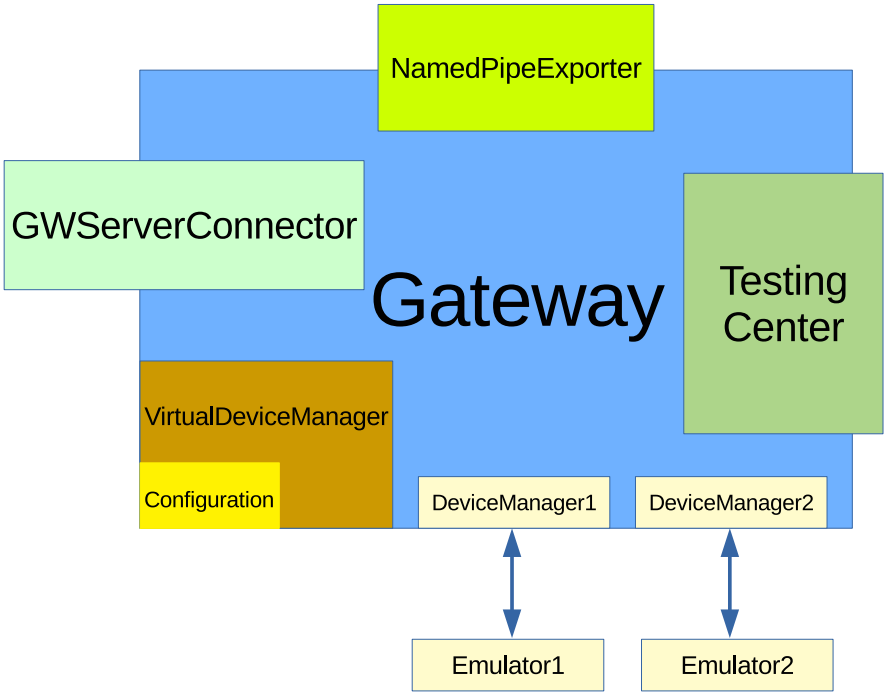
Po bokoch sú ďalej znázornené možnosti pre vstup riadiacich príkazov. Jedná sa o pomerne invazívne využitie prvku Testovacieho centra a neinvazívne pripojením na testovacím prostredím emulovaný server.

Na vrchu sa nachádza export senzorických dát, ktorý môže mať rôzne implementácie. Pri integračnom testovaní bude použitá implementácia pomocou pomenovanej rúry (*Named Pipe*) v textovom formáte JSON a export na Gateway Server, ktorý rovnako využíva formát JSON.

Integráciou aplikácie brány budeme testovať nasledujúce scenáre:

- Export dát – Aplikácia so špecifikovanými napárovanými virtuálnymi senzormi by mala periodicky exportovať dáta, ktoré by mali byť viditeľné na všetkých exportéroch.
- Párovanie a pridanie zariadenia – Aplikácia s definovanými zariadeniami, ktoré nie sú predom napárované nesmie exportovať žiadne dáta kým neprebehne celý proces párovania popísaný v predchádzajúcej kapitole. Po prebehnutí párovacieho procesu sa opäť testuje export dát.

- Modifikácia aktívneho prvku – Pri systéme s definovaným aktívnym virtuálnym prvkom musí prejsť proces nastavenia hodnoty s výsledkom, ktorý bol predom definovaný.



Obr. 4.1: Vonkajšie rozhranie aplikácie brány

4.1.2 Testovanie GW Server

Rozhranie časti serveru zodpovednú za komunikáciu s bránami je možné vidieť na obrázku 4.2. Pre komunikáciu s bránami je použité rozhranie WebSocket na najvyššej vrstve. Na spodku aplikácie sa nachádza dátová vrstva, ktorá využíva databázu. Testovacia databáza musí byť vždy inicializovaná pre konkrétny účel testu. Testovacie scenáre:

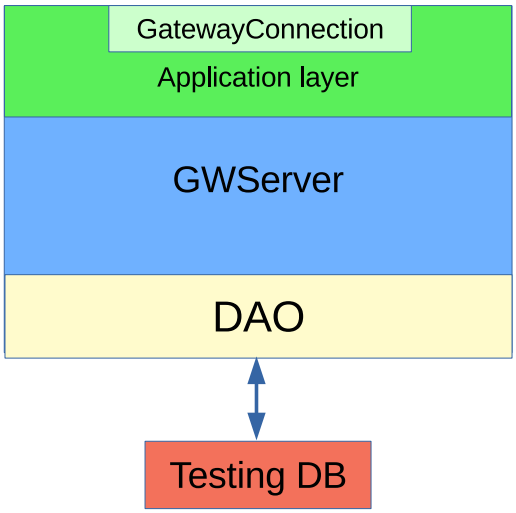
- Export dát – Server prijíma dátové správy z brány a ukladá ich do databázy.
- Správa brány – Registrácia brány v systéme po jej pripojení a preposielanie požiadaviek.

4.1.3 Testovanie UI Serveru

Pre interakciu s touto časťou systému pre testovanie môžeme využiť oba druhy rozhraní, ktoré ponúka. Ide teda o komunikáciu s vyuzitím technológie REST a s využitím protokolu na báze XML tak, ako je to možné vidieť na obrázku 4.3.

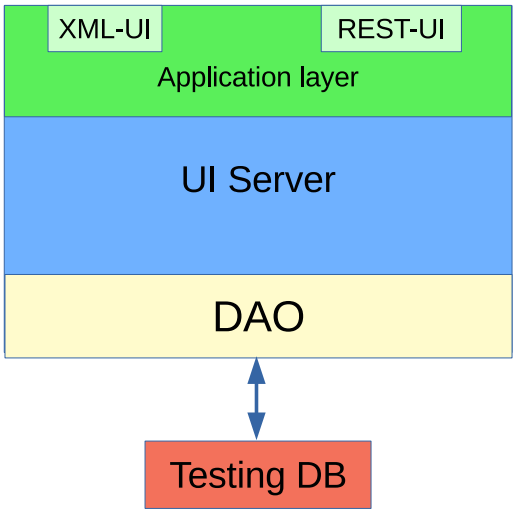
Integráciou tejto časi budeme testovať nasledujúce scenáre, ktoré môžeme rozdeliť do 3 kategórií:

- Správa používateľov – Registrácia, autorizácia a prístup k detailom profilu.



Obr. 4.2: Vonkajšie rozhranie GWServer

- Správa brán pre používateľa – Registrácia brány, získanie používateľových brán, inicializácia párovacieho režimu.
- Správa zariadení – Získanie zariadení brány, pridanie a odobranie zariadení, získanie poslednej meranej hodnoty a historických dát, modifikácia stavu aktívneho prvku.



Obr. 4.3: Vonkajšie rozhranie UIServer

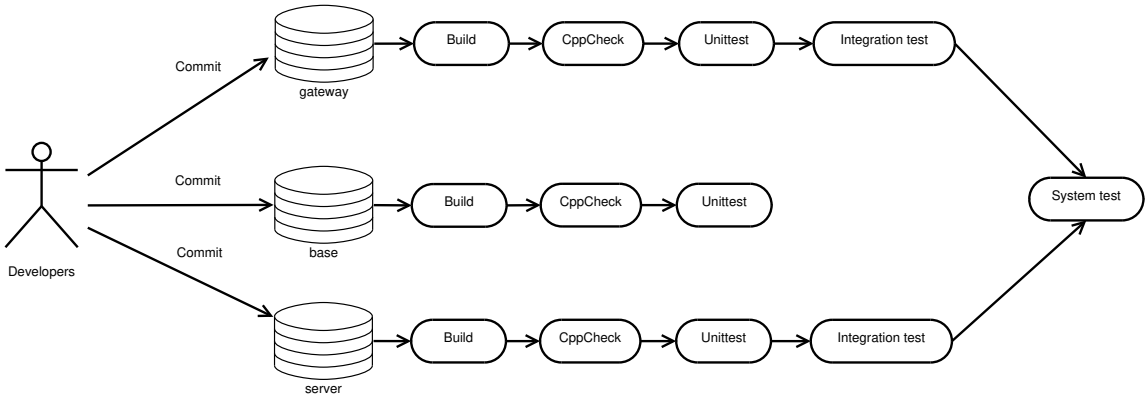
4.2 Systémové testy

V prípade systémových testov sú prítomné všetky spomenuté komponenty a budú testované scenáre naprieč celým systémom. Testované budú scenáre, ktoré vyžadujú interakciu medzi serverom a bránou:

- Registrácia spustenej brány v systéme a asociácia s používateľom.
- Exportovanie dát virtuálneho zariadenia a získavanie týchto dát používateľom.
- Párovací proces pre virtuálne zariadenie. Overíme tu, či bolo dané zariadenie nájdené počas párovacieho režimu a správne reportované, následne ho prijmeme a odpárujeme.
- Nastavovanie hodnoty virtuálneho zariadenia.

4.3 Integrácia do Jenkins CI

Pretože je v projekte už dlhšie využívaný nástroj pre Kontinuálnu integráciu *Jenkins* rozhodol som sa ho použiť pre automatizáciu testovania. Najdôležitejšou časťou je návrh zreťazenej linky (*pipeline*) reprezentovaný na obrázku 4.4. Celý proces začína vytvorením commitu do centrálneho repozitára projektu. V závislosti od repozitára, pre ktorý bol commit vytvorený sa konkrétny projekt preloží, spustí sa statická analýza kódu a jednotkové testy. V prípade repozitára brány a serveru sa vykoná integračný test danej súčasti. Na záver sa systémovým testom potvrdí zachovanie celkovej stability systému v oboch prípadoch.



Obr. 4.4: Návrh zreťazenej linky Kontinuálnej integrácie

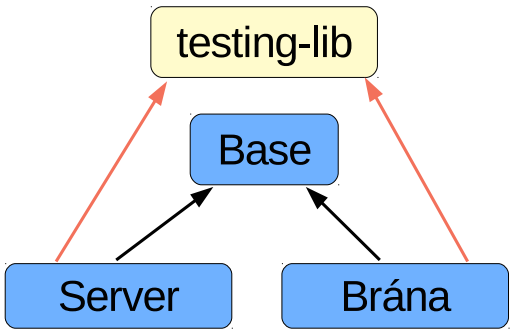
Kapitola 5

Implementácia

Pre implementáciu bolo potrebné v prvom rade zvoliť vhodný jazyk a použité prostredie. Pre jednotkové testy, ako je to typické, je použitý natívny jazyk aplikácie, teda jazyk C++ s využitím knižnice CppUnit. V prípade integračných a systémových testov je však vhodné použit nejaký zo skriptovacích jazykov, ako je napríklad Perl alebo Python. Pre tento účel som zvolil jazyk Python 3 z dôvodu bohatosti štandardne dostupných knižníc, jeho dostupnosti, jednoduchosti použitia a mojich osobných skúseností s týmto jazykom. Ako základ použijem knižnicu Python unittest. Napriek názvu tejto knižnice, ktorý napovedá, že slúži pre tvorbu jednotkových testov, poskytuje vhodné nástroje, techniky a koncepcie použiteľné aj pre testy vyšších úrovní.

5.1 Umiestnenie zdrojových kódov

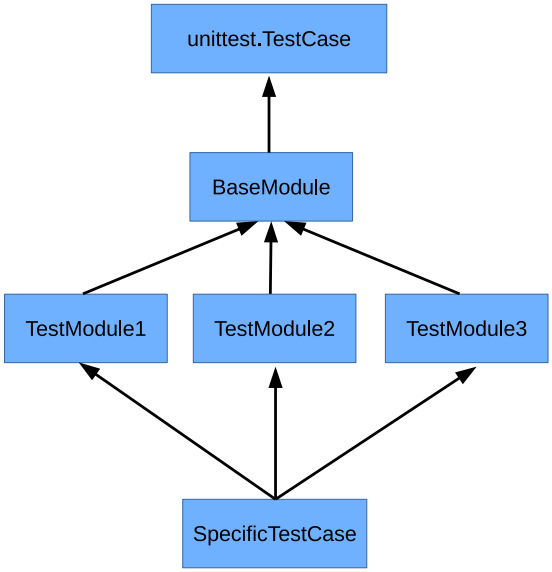
Okrem iného je tiež dôležité zvoliť a jasne si definovať, kde sa nachádzajú zdrojové kódy pre automatické testovanie. Pre základné testovacie štruktúry, pomocný kód a utility, nakoľko môžu byť zdielané naprieč testovaním integrácie viacerých častí bol vytvorený samostatný git repozitár s názvom testing-lib. Samotné testovacie scenáre pre integračné testy sú potom súčasťou už existujúcich repozitárov. Potrebný repozitár testing-lib je potom zahrnutý formou podmodulu verzovacieho systému Git. Vzťahy medzi týmito repozitármi a novovytvorený repozitár je možné vidieť na obrázku 5.1, pričom novovzniknutý repozitár je farebne odlíšený.



Obr. 5.1: Vzťahy medzi repozitármi a novovytvorený repozitár testing-lib.

5.2 Modulárne testy

Modulárnosť samotného testovaného systému (najmä aplikácie brány) je potrebné reflekovať aj implementovaným testovacím prostredím, ktoré tiež musí byť spravené v rámci možnosti modulárne.



Obr. 5.2: Koncept testovacej architektúry

Na obrázku 5.2 možno vidieť koncept implementácie testovacej architektúry. Samotný konkrétny testovací scenár (SpecificTestCase) nachádzajúci sa v spodnej časti obrázku definuje priebeh jednotlivých testov. Tento testovací scenár využíva ľubovolné množstvo testovacích modulov (TestModule), pričom každý takýto testovací modul implementuje prístup k nejakej špecifickej časti systému. Môže sa jednať napríklad o emulátor nejakého špecifického zariadenia, jednoduchú emuláciu serveru a podobne. Testovací modul je naviac zodpovedný za inicializáciu a deinicializáciu danej časti, či automatická úprava konfigurácie samotnej spúštanej aplikácie, pokiaľ je to potrebné. Pre tento účel môže byť využitý konštruktor triedy, alebo metódy `setUp()` a `tearDown()`, ktoré sú volané automaticky testovacím prostredím unittest. Všetky moduly majú spoločného predka BaseTest, ktorý má za úlohu samotné spustenie testovanej aplikácie v metóde `setUp()`, takže máme zaistené, že samotný test je spustený až po inicializácii všetkých modulov a bežiacej testovanej aplikácie. BaseModule je nakoniec rozšírením triedy `unittest.TestCase`, takže s ním môže testovací systém pracovat genericky.

Využijeme tu možnosť viacnásobnej dedičnosti jazyka Python, kedy špecifický testovací scenár je potomkom všetkých využitých modulov. Aby sme postupne inicializovali a deinicializovali všetky použité moduly, je dôležité aby každý modul explicitne volal `setUp()` a `tearDown()` tú istú metódu svojeho predka tak, ako je to na ukážke 5.1. V prípade `tearDown()` je dôležité zavolať metódu svojeho predka aj v prípade vyskytnutej výnimky, pretože by po teste mohol bežať, napríklad neukončený proces, za ktorý sme zodpovední. Pretože sú moduly inicializované vzostupne od špecifického k všeobecnému, je možné aby špecifický testovací scenár dodatočne upravil konfiguráciu alebo vlasnosti nadradeného modulu. Tento princíp nám tiež zaručuje, že samotná aplikácia, ktorá je súčasťou hlavného modulu sa spustí až ako posledná po inicializácii ostatných modulov.

```
class TestModule(BaseModule):
        def setUp(self):
                # initialise
                super().setUp()

        def tearDown(self):
                try:
                        # cleanup
                finally:
                        super().tearDown()
```

Výpis 5.1: Ukážka implementácie testovacieho modulu.

5.2.1 ApplicationRunner

Pre účel kontrolovaného spúšťania aplikácií bola implementovaná trieda `ApplicationRunner`.

Konštruktor triedy má päť argumentov:

- `binaryPath` – Cesta k binárnemu súboru spúšťanej aplikácie.
- `configPath` – Cesta k hlavnému používateľskému súboru aplikácie.
- `stderrFile` – Cesta k súboru, do ktorého má byť presmerovaný štandardný chybový výstup aplikácie.
- `stdoutFile` – Cesta k súboru, do ktorého má byť presmerovaný štandardný výstup aplikácie.
- `valgrind` – Voliteľná možnosť spustit aplikáciu s využitím programu valgrind, pre detekciu pamäťových únikov.

Spustenie aplikácie prebehne po volaní metódy `start()`, kedy pomocou systémových volaní `fork()` a `execv()` vytvorí podproces, v ktorom bude bežat samotná aplikácia. `ApplicationRunner` využíva parameter `-N`, ktorý je vysvetlený v podkapitole 3.8.3. Pri spúšťaní samotnej aplikácie jej špecifikuje tento prepínač s vlastným číslom procesu. Po volaní `fork()` potom očakáva obdržanie signálnu `SIGTERM`. Pred volaním `execv()` sú naviac presmerované deskriptory `stdout` a `stderr` do otvoreného súboru, ktorý vieme po prebehnutí testov archivovať a v prípade problematického testu poskytnúť vývojárovi pre pomoc pri ladení problému.

Zastavenie rozbehnutého procesu poskytuje instančná metóda `stop()` triedy `ApplicationRunner`. Spustenému podprocesu pošleme signál typu `SIGTERM` a štandardne čakáme na ukončenie potomka a prevzatím jeho návratovej hodnoty. V prípade, že nedôjde k ukončeniu aplikácie v určitom časovom limite, je vygenerovaný signál typu `SIGKILL`, ktorý už nie je možné ignorovať. Metóda `stop()` vracia návratovú hodnotu ukončeného podprocesu, ktorá by mala byť, v prípade korektného ukončenia nulová.

`ApplicationRunner` okrem tohoto podporuje aj vynútenú modifikáciu používateľskej konfigurácie s využitím parametru `-D`, ktorý je vysvetlený v podkapitole 3.8.3. Instančnou metódou `overrideUpdate(key, value)` triedy `ApplicationRunner` vieme vynútiť. Ako príklad, znázornený v ukážke 5.2 si uvedieme spustenie aplikácie brány s podporou serveru a pripojením na port 1010 a ip adresu `172.16.0.1`.

```
from ApplicationRunner import ApplicationRunner

runner = ApplicationRunner("./beeeon-gateway",
               "./conf/gateway-startup.ini",
               "gateway.err",
               "gateway.out")
runner.overrideUpdate("gws.enable", "yes")
runner.overrideUpdate("gws.host", "172.16.0.1")
runner.overrideUpdate("gws.port", "1010")

runner.start()

runner.stop(10)
```

Výpis 5.2: Ukážkové použitie triedy `ApplicationRunner`.

5.2.2 GatewayBaseModule a ServerBaseModule

Hlavnou úlohou týchto tried je inicializácia a spustenie testovanej aplikácii brány (`GatewayBaseModule`) a serveru (`ServerBaseModule`). Testované aplikácie bežia s využitím základných predvolených konfiguračných XML súborov pre *Depdendency Injector*, čím súčasne overujeme validitu tejto konfigurácie. Z pohľadu používateľskej konfigurácie využijeme tak isto dostupné a udržované testovacie konfiguračné súbory, ktoré sa vyznačujú hlavne tým, že majú, čo možno najviac dostupných komponentov implicitne vypnutých. Tým pádom si pre testovací účel vieme zapnúť a nastaviť len tie komponenty, ktoré skutočne potrebujeme. Spúšťanie prebieha s využitím triedy `ApplicationRunner`, ktorá je špecifikovaná v podkapitole 5.2.1.

5.3 Problémy dĺžky testov

Zaujímavý problém, ktorý som počas implementácie riešil bol časový interval priebehu testovacích sád. Inicializácia a deinicializácia každej testovacej sady trvá určitý čas. Prispieva k tomu napríklad spúšťanie a inicializácia samotnej aplikácie, jej deinicializácia a ukončenie ako aj vytvorenie ďalších potrebných nástrojov. Len tieto úkony môžu trvať jednotky až desiatky sekúnd, čo sa pri stovkách testovacích sád dosť výrazne prejavuje. Naviac, niektoré testy si zo svojej povahy vyžadujú explicitne čakať určitý čas. Napríklad pre overenie, že pripojená aplikácia drží spojenie musíme explicitne čakať určitý čas a overiť, že sa daná aplikácia neočakávane neodpojila. Iný príklad môže byť dokončenie nejakého evokovaného procesu, napríklad príkaz *listen* v sebe obsahuje parameter dĺžky v sekundách, ktorý určuje ako dlho má vyhľadávanie nových zariadení prebiehať. Pokiaľ pošleme príkaz *listen* s dĺžkou 10 sekúnd, musíme v teste čakať aspoň 10 sekúnd ak chceme skontrolovať akýkoľvek stav aplikácie po vykonaní tohoto príkazu a súčasne nepoužívať žiadne invazívne nástroje. Riešením je pri takýchto testoch zvoliť nejaký rozumný kompromis a v konkrétnych prípadoch voliť také hodnoty, ktoré výrazne neovplyvnia celkovú dĺžku testov a súčasne sa vyhnúť extrémom, ktoré by znehodnotili výsledok testovania.

S problémom dĺžky testov súvisí opakované testovanie, ktoré tento problém ešte znásobuje. Pretože BeeeOn aplikácie fungujú vysoko asynchrónne s množstvom aktívnych vlákien, niektoré problémy spôsobené takzvaným *Race Condition* sa nemusia deterministicky vždy prejaviť. Ani opakované testy nezaručujú odhalenie takéhoto prípadu, no z praktického

hľadiska výrazne zvyšujú šancu na jeho prejavenie a v konečnom dôsledku vedú k zlepšeniu kvality testovaného softvéru. V spojení s problémom dĺžky testov môže rozumné množstvo opakovaných testov prebiehať aj niekoľko desiatok minút, či jednotky hodín.

5.4 Cppcheck

Použitie nástroju Cppcheck pre statickú analýzu kódu je celkom priamočiare a je možné ho použiť v ktoromkoľvek repozitáry projektu nasledovne:

```
$ cppcheck -enable=all -inconclusive src 2> cppReport
```

Použijeme teda všetky dostupné testy tohoto nástroja statickej analýzy kódu. Výsledok takejto analýzy je vždy potrebné preveriť, nakoľko sa môže jednať aj o *False positive* výsledky, teda také, ktoré nesprávne indikujú chybu. Filozofia tohoto projektu je však taká, aby bolo takýchto falošných výsledkov čo najmenej, napriek tomu sa to stať môže [2]. Výsledok sa po analýze nachádza v súbore „c“ppReport.

5.5 Integračné testy brány

Brána je najmodulárnejšia časť systému. V tejto podkapitole sa budem venovať jednotlivým testovacím modulom, ktoré sa tu využívajú a samotným testovacím scenárom.

5.5.1 FakeGWServerModule

Pre emuláciu GW Serveru, na ktorý sa pripája brána prostredníctvom protokolu WebSocket existuje modul `FakeGWServerModule`. Tento modul konfiguruje aplikáciu tak, aby sa pripájala na vopred stanovený port v rámci lokálneho stroja. Následne vytvorí inštanciu `FakeGWServer`, ktorá emuluje skutočný server vrátane protokolu a počúva na danom porte.

FakeGWServer

Základné knižnice jazyka Python 3 bohužial neimplementujú WebSocket protokol a pre vznik tohoto nástroja bolo potrebné použiť knižnicu tretej strany. Pre tento účel sú dostupné dva projekty:

- „tornado.websocket“ [1] – Pomerne komplexná knižnica so zložitejším API a optimalizáciou pre reálne nasaditeľné aplikácie, ktoré zvládajú asynchronne spracovávať viaceré spojenia.
- „python-websocket-server“ [2] – Jednoduchá implementácia protokolu WebSocket, ktorá nie je až taká komplexná, ale je vhodnejšia pre toto použitie z dôvodu jej jednoduchosti.

`FakeGWServer` je špeciálna trieda, ktorá emuluje skutočný Gateway Server s obmedzením na maximálne jedného klienta, ktorého práve testujeme. Celá funkcionalita je implementovaná asynchronne v samostatnom vlákne. Okrem konfigurácie portu, na ktorom má naslúchať WebSocket protokol, obsahuje aj rozšírenú konfiguráciu, ktorá určuje behaviorálne správanie. Táto konfigurácia sa skladá z nasledovných parametrov:

[1] http://www.tornadoweb.org/en/stable/websocket.html
[2] https://github.com/Pithikos/python-websocket-server

- autoRegister – určuje, či majú byť registračné správy brány automaticky potvrdené príslušnou potvrdzovacou správou.
- autoResponse – Nastavenie tohoto atribútu spôsobuje automatické spracovanie odpovedí z brány. V prípade, že takáto odpoveď vyžaduje potvrdenie, je vygenerované a poslané bráne. Jedná sa o správy typu *generic_response*, ktorá nevyžaduje odpoveď a *response_with_ack*, ktorá vyžaduje odpoveď.
- autoData – Ide o automatické spracovanie dátových správ, pričom sa rovnako vygeneruje potvrdzovacia správa s príslušným identifikačným číslom správy. Takáto dátová správa je spracovaná a uložená do fronty prijatých dátových správ. Ide o správy typu *sensor_data_export*.
- prePaired – Zoznam zariadení, ktoré sú predpárované. Brána sa po svojom štarte typicky dotazuje serveru, ktoré zariadenia jej patria a server na takúto správu odpovedá zoznamom, kde sa nachádzajú id zariadení, ktoré sú z pohľadu serveru napárované.
- autoNewDevice – Automatické spracovanie správ typu *new_device_command*.

Tieto rozširujúce atribúty sú typicky vypnuté pri testovaní samotnej funkcionality. Pri testovaní komplexnejšej funkcionality sú naopak aktivované pre zjednodušenie testovania a možnosti zamerať sa na testovanú funkcionalitu.

Pre riadenie príkazov tu existuje trieda `Commander`, ktorá dokáže automaticky riadiť priebeh príkazu pre bránu. Daný príkaz vygeneruje so špecifikovanými hodnotami, je odoslaný pripojenej testovanej aplikácii a následne čaká na jeho spracovanie a finálnu odpoveď. Toto čakanie je s časovým limitom, ktorý pokiaľ vyprší, je operácia považovaná za zlyhanú. Po obdržaní finálnej odpovedi je potom výsledok operácie propagovaný vyššie, teda do samotného testovacieho scenáru. Podporované operácie sú `listen()`, `deviceAccept()`, `unpair()` a `deviceSetValue()`.

5.5.2 TCModule

V predchádzajúcej kapitole 3.3.5 bola špecifikovaná komponenta aplikácie brány Testovacie centrum. Táto časť systému umožňuje do aplikácie posielať rôzne príkazy a emulovať určité odpovede serveru. Testovací modul TCModule je zodpovedný za konfiguráciu aplikácie tak, aby bolo Testovacie centrum aktivované a počúvalo na predom špecifikovanom porte. Okrem toho aj obaluje celú komunikáciu s Testovacím centrom, ktorá prebieha prostredníctvom klasického TCP socketu. Podobne ako pri `GWServerModule` tu existuje `Commander`, ktorý navonok funguje prakticky totožne. Riadi vykonávanie nejakého príkazu so špecifickými parametrami a časovým limitom. Na rozdiel od `GWServerModule` tu získavame detailnejšiu odpoveď, pretože máme informácie koľko komponentov danú správu prijalo, koľko z nich skončilo v stave `SUCCESS` a koľko z nich v stave `FAILED`. Pri odpovedi na server sa tieto hodnoty v súčasnej verzii agregujú do jednotného stavu.

5.5.3 NamedPipeModule

Tento testovací modul poskytuje podporu pre export dát aplikácie brány prostredníctvom pomenovanej rúry. Predovšetkým konfiguruje aplikáciu brány so zvolenou cestou k súboru, ktorý predstavuje pomenovanú rúru. Rovnako inicializuje komponentu, ktorá z tohoto súboru načítava dáta a dokáže ich spracovať na základe stanoveného formátu. Špecifický test

potom môže overiť napríklad funkčnosť exportu samotných dát. Spracovanie dát prebieha asynchronne v samostatnom vlákne, pričom sú spracované dáta ukladané do fronty.

5.5.4 VdevModule

Jedná sa o testovací modul, ktorý poskytuje podporu pre použitie komponenty virtuálnych zariadení brány pre testovacie účely. Okrem explicitného povolenia virtuálnych zariadení pre aplikáciu brány je jeho zodpovednosť aj vytvorenie dočasného konfiguračného súboru pre definíciu potrebných virtuálnych zariadení na základe instančného atribútu, ktorý je nastavený v špecifickom testovacom scenári. V ukážke 5.3 je možné vidieť príklad použitia tejto triedy.

```
vdevIni =
"""
[virtual-devices]
request.device_list = yes

[virtual-device1]
enable = yes
paired = yes
refresh = 60
device_id = 0xa300000000000002
vendor = VUT / RehiveTech
product = Air pressure sensor
module0.type = pressure
module0.min = 0
module0.max = 15
module0.generator = random
module0.reaction = none
"""
class TestVdevExample(VdevModule):
        def setUp(self):
                self.vdevConfig = vdevIni
                super().setUp()
```

Výpis 5.3: Príklad použitia `VdevModule`.

5.5.5 Testovacie scenáre

Testy pre bránu vieme rozdeliť do niekoľkých kategórií:

- core – V testoch jadra sa snažíme testovať dostupné moduly aplikácie bez znalosti ich samotného fungovania. Tieto testy sú natoľko všeobecné, že k ich fungovaniu nám pomáha šablóna, ktorú zdedíme pri testovaní. Kontrolujeme tu predovšekým reakcie na všeobecné príkazy, kontrolujeme či neostanú visieť nejaké odpovede, či nejaký modul neprijíma príkazy, ktoré mu nepatria, respektíve vždy prijme príkaz, ktorý mu patrí. Tieto testy sú vykonané postupne všetkými izolovanými modulmi. Na konci sa pustí aplikácia so všetkými dostupnými komponentami systému a opäť sa kontroluje korektnosť spracovania príkazov.

- vdev – S využitím Testovacieho Centra otestujeme funkčnosť virtuálnych zariadení. Testujú sa typické scenáre ako párovanie, odpárovanie, export dát, alebo žiadosť serveru o napárované zariadenia.

- gws – Čiastočne s použitím virtuálnych senzorov otestujeme serverový konektor, pričom simulujeme klasické prípady použitia, ale aj rôzne neočakávané správy, ktoré by mali byť korektne nasledované ukončením spojenia zo strany brány.

- psdev – Táto testovacia sada je zameraná na testovanie správcu zariadení, ktorý je zodpovedný za skutočné zariadenie. Jedná sa o senzor atmosferického tlaku, ktorý je vstavaný v hardvéri BeeeOn brány. Pre tohoto správcu nie je potrebný žiadny zložitý emulátor, pretože načítavanie senzorických dát je prostredníctvom súboru v súborovom systéme. Takýto súbor je jednoduché podstrčiť.

- belkin – Vzhľadom na existenciu emulátorov zariadení typu Belkin WeMo môžeme s ich využitím otestovať aj správcu týchto zariadení s využitím prítomných emulátorov. Testované budú scenáre ako vyhľadávanie zariadení, párovací proces, export dát a nastavovanie aktívnych prvkov.

5.6 Integračné testy serveru

Hoci projekt serveru obsahuje sadu funkčných testov, tie nie sú plne automatizované, pretože už predpokladajú, že aplikácia, databázový server a iné závislosti už bežia a neriešia teda ich inicializáciu. Vzhľadom na modulárnosť implementovaného testovacieho prostredia a toho, že funkčné testy serveru sú písané v jazyku Python 3 a využitím knižnice *Python.unittest*, ich úprava pre plnú automatizáciu bola viac menej jednoduchá. Miesto priameho dedenia triedy `unittest.TestCase`, museli všetky implementované testy dediť testovacie moduly, ktoré zabezpečujú všetky potrebné závislosti. Testovacie scenáre serveru sú rozdelené do 3 kategórií:

- gws – Testy zamerané na komponentu GW Server, kde sa na server pripájame prostredníctvom WebSocket rozhrania. Testuje sa tu základná konektivita, spracovanie riadiacich rámcov protokolu WebSocket a samotné správy aplikačného protokolu, napríklad registračná správa, reportovanie nových zariadení, export dát a podobne.

- restui – Testy zamerané na podčasť REST-UI komponenty UI Server. Konkrétne testujeme používateľské rozhranie implementované s využitím technológie REST. Tak isto sa tu overuje základná konektivita, funkčnosť autorizácie, správa používateľov, brán, zariadení a v neposlednom rade aj dotazovanie sa na senzorické dáta.

- xmlui – Charakterovo sú tieto testy podobné ako *restui*, s tým rozdielom, že tu testujeme rozhranie so špecifickým aplikačným protokolom na báze XML. V dobe písania tejto práce je tento protokol udržiavaný z hlavne kompatibilných dôvodov voči existujúcej mobilnej aplikácii BeeeOn pre operačný systém Android, ktorá ešte nebola upravená pre použitie REST-UI.

Všetky testovacie sady prebiehajú nad databázou, ktorá je aplikáciou najskôr inicializovaná špecifickými testovacími dátami.

5.6.1 DatabaseModule

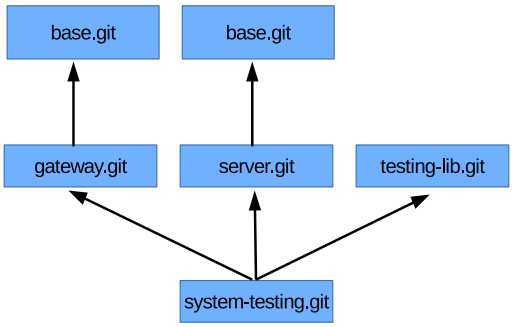
Databázový modul, ktorý má za úlohu vytvoriť dočasnú databázu v privátnej inštancii PostgreSQL serveru za účelom behu testov. Okrem toho aj nakonfiguruje samotnú aplikáciu serveru tak, aby sa mohla na túto databázu pripojiť. Pre implementáciu bola použitá knižnica *testing.postgresql* [5], ktorá je presne určená pre takéto použitie.

5.6.2 RestModule

Tento pomocný modul zabezpečuje pripojenie a autorizáciu pre UI server prostredníctvom Rest API. Tak isto konfiguruje aplikáciu serveru tak, aby Rest API bežalo na špecifickom porte, na ktorý sa môžeme pripojiť. Ponúka tiež metódy, ktoré abstrahujú samotnú komunikáciu pre prehladnejšie a jednoduchšie popisy testovacích prípadov.

5.7 Systémové testy

Podstata systémových testov je integrácia softvéru brány so softvérom serveru. Pretože oba testované celky majú svoj samostatný verzovací repozitár, testovacie scenáre sa nemôžu nachádzať ani v jednom z nich. Pre tento účel teda musel byť vytvorený nový, samostatný repozitár „system-testing“. Tento repozitár obsahuje 3 samostatné podmoduly: „gateway.git“ pre kód brány, „server.git“ pre kód serveru a „testing-lib.git“ pre pomocnú testovaciu knižnicu, ktorá sa používa aj v integračných testoch. Pretože repozitáre brány aj serveru majú ešte samy o sebe podmodul repozitáru „base.git“, musíme ich pred testovaním tiež inicializovať. Celá štruktúra je graficky znázornená na obrázku 5.3. Repozitáre „base.git“ sú tu zámerne reprezentované samostatne, pretože, hoci sa jedná o ten istý repozitár, môže ísť v tomto prípade o iné revízie (a typicky aj ide). Pre vykonanie systémových testov je tiež dôležité špecifikovať, ktorá revízia softvéru brány a serveru sa má použiť. Rôzne kombinácie nemusia byť nutne kompatibilné. Typicky by však mala byť zachovaná kompatibilita medzi hlavnými (vetvami „master“) revíziami.



Obr. 5.3: Štruktúra podmodulov systémových testov.

5.7.1 Testovacie scenáre

Vzhľadom na to, že pri systémových testoch testujeme hlavne integráciu brány a serveru, testovacie scenáre obmedzíme na tie, v ktorých vystupujú obe tieto nezávislé komponenty. Jedná sa o tieto 4 scenáre:

- Registrácia – Pri teste registrácie sa brána zaregistruje na server, následne ju používateľ priradí k svojmu používateľskému účtu.
- Párovací proces – S využitím virtuálnych zariadení na bráne môžeme overiť funkčnosť párovacieho procesu. V tomto teste disponuje brána s virtuálnym zariadením, ktoré má k dispozícii a je nenapárované. Používateľ dá príkaz vyhľadať nové zariadenia na tejto bráne, pri čom by sa malo virtuálne zariadenie správne reportovať. Používateľ zariadenie aktivuje, čím sa po úspešnej odpovedi z brány zariadenie stáva napárované. Po celom tomto procese zariadenie ešte odpárujeme a skontrolujeme, či všetko prebehlo podľa očakávaní.
- Export dát – Pri tomto teste napárujeme k bráne virtuálne zariadenie, ktoré generuje hodnoty, overíme, či sú hodnoty reportované správne.
- Natavenie hodnoty – V tomto scenári si napárujeme dve virtuálne zariadenia s aktívnymi prvkami. Nastavenie jedného máme definované ako úspech a druhé ako zlyhanie. Obe sa pokúsime nastaviť.

5.8 Integrácia do Jenkins CI

Integrácia do systému Jenkins CI. Nakoľko repozitáre brány a serveru sú v projekte zaintegrované do tohoto systému pre kompiláciu a jednotkové testy, je potrebné pomocou Jenkins Pipeline skriptu pridať fázu pre integračné testy a statickú analýzu kódu. Kód tejto fázy pre integračné testy môžeme vidieť v ukážke 5.4.

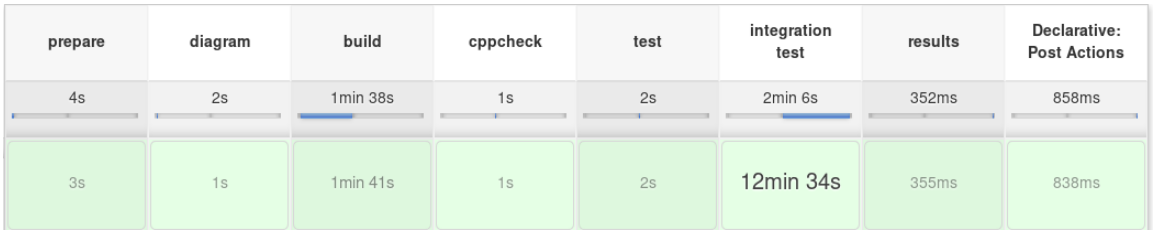
```
stage('integration test') {
        steps {
                script {
                        if (fileExists('t/testAll.sh')) {
                                echo "Testing"
                                sh 'cd t;./testAll.sh'
                        } else {
                                echo "Not testing"
                        }
                }
        }
}
```

Výpis 5.4: Popis fázy integračných testov pre Jenkins CI.

Môžme si všimnúť, že spúštanie integračných testov musí byť podmienené existenciou skriptu „testAll.sh“, ktorý indikuje, že sa integračné testy v revízii nachádzajú. Táto podmienka musí existovať do vtedy kým nebude kód testov začlenený do všetých používaných vetví verzovacieho systému. Graficky znázornený priebeh celej zreťazenej linky pre integráciu aplikácie brány je možné vidieť na snímke obrazovky 5.4, vidíme priebeh jednotlivých fáz, od kompilácie, jednotkových testov, automatickej statickej analýzy a integračných testov, až po vyhodnotenie výsledku úlohy a dodatočné akcie.

Systémové testovanie

Pre systémové testovanie bola vytvorená samostatná úloha pre Jenkins CI, ktorá je zložená len z troch fáz: inicializácia, test a vyhodnotenie. Táto úloha je parametrizovateľná



Obr. 5.4: Priebeh zreťazenej linky pre integráciu aplikácie brány.

a obsahuje tri parametre: revíziu pre repozitár brány, revízu pre repozitár serveru a počet opakovaní, ktoré sa majú vykonať. Túto úlohu je možné spustiť manuálne, alebo je možné ju invokovať ukončením nejakej inej úlohy (napr. integráciou aplikácie brány/serveru). Na obrázku 5.5 môžeme vidieť grafické znázornenie zreťazenej linky Jenkins pre systémové testy. Test bol najskôr spustený s jediným opakovaním, následne, bol spustený s parametrom opakovania 10. Tu je možné vidieť lineárnu časovú závislosť dĺžke testov a ich opakovaniu.

Obr. 5.5: Priebeh zreťazenej linky pre systémové testy.

Kapitola 6

Výsledky a metriky

Navrhnuté riešenie automatických testov sa mi podarilo úspešne implementovať a integrovať do vývoja projektu BeeeOn. V tejto záverečnej kapitole sa budem venovať niektorým odhaleným problémom a zhrniem objektívne metriky testovania.

6.1 Odhalené problémy

Napriek regresnej povahe implementovaných testov, fakt, že rôzne časti projektu neboli priebežne funkčne testované (najmä softvér brány) odhalilo problémy aj v súčasnej verzii. V tejto podkapitole by som sa chcel venovať pár zaujímavým odhaleným problémom. Všetky nájdené problémy som buď následne sám opravil, alebo reportoval vývojovému tímu na opravu.

Zdielaná inštancia triedy Poco::Event

Tento problém je zložitejší než sa na prvý pohľad zdá. Jedná sa o *race condition* pri spracovávaní odpovedí na príkazy v aplikácii brány a využití synchronizačného nástroja typu udalosť (Event).

Každý objekt, ktorý v systéme dokáže odosielať príkazy obsahuje vlastnú frontu určenú pre odpovede k odoslaným príkazom. Táto fronta obsahovala zdieľanú udalosť, ktorá informovala o akejkoľvek zmene, ktoréhokoľvek príkazu vo fronte, takže dokázala prebudiť vlákno čakajúce na nejakú odpoveď.

Problém nastal, ak na jednu udalosť čakalo viacero vlákien na ukončenie rôznych príkazov. Ak totiž prišla notifikácia o ukončení nejakého príkazu, nedeterministicky vzbudila len jedno z čakajúcich vlákien. V prípade, že sa vzbudilo vlákno, ktorého príkaz ešte nie je ukončený, toto vlákno sa znovu uspalo na udalosť, to viedlo až k hladoveniu a uviaznutiu čakajúceho vlákna.

Nakoľko je takáto situácia ojedinelá v reálnom prostredí, odhalili ju až automatické testy, ktoré museli byť viacnásobne opakované, pretože ani v testovacom prostredí sa to nemusí prejaviť vždy.

Race condition v GWServerConnector

Zaujímavý problém odhalili aj integračné testy zamerané na serverový komunikátor aplikácie brány. GWServerConnector funguje s využitím dvoch vlákien, pričom jedno vlákno prijíma správy a druhé ich naopak odosiela. Niektoré typy správ vyžadujú od servera po-

tvrdzujúcu odpoveď, v takýchto prípadoch odosielacie vlákno správu odošle a naplánuje jej znovuodoslanie v prípade, že by neprišla odpoveď do určitého časového limitu.

Problém je krátke časové okno medzi odoslaním správy a jej naplánovaním, počas ktorého môže pri veľmi nízkej latencii servera prísť odpoveď, ktorá je spracovaná a úloha znovuodoslania zrušená ešte pred tým, než je naplánovaná, to vedie k nelegálnemu stavu aplikácie a v lepšom prípade len k odpojeniu od servera a jeho znovupripojeniu. V reálnom prostredí by bola takáto situácia zriedkavá a prakticky nereprodukovateľná. Pri automatických testoch na jednom stroji je však latencia takmer nulová, čo takmer zaručuje prejavenie takéhoto správania.

Podozrivá ukazovateľová aritmetika

Tento príklad som sa sem rozhodol zahrnúť, pretože bol objavený automatickou statickou analýzou kódu v repozitáry BeeeOn Base. Problematický bol kus kódu v ukážke

```
for (const char c : input) {
        ...
        throw SyntaxException("unfinished escape character on: " + c);
        ...
}
```

Výpis 6.1: Ukážka problému odhaleného statickou analýzou.

Autor tohoto kódu chcel očividne spraviť konkatenáciu reťazca so znakom. Pretože ale v jazyku C++ je reťazcový literál reprezentovaný ako ukazateľ do konštantnej pamäte, sémanticky sa jedná o súčet ukazateľov. Vykonanie takéhoto kódu je nebezpečné a môže viesť k pretečeniu pridelenej pamäte a eventuélne pádu celej aplikácie. Nástroj Cppcheck na toto upozornil takýmto spôsobom: „`(error) Unusual pointer arithmetic. A value of type 'char' is added to a string literal.`"

6.2 Metriky testovaného softvéru

Pre predstavu komplexnosti testovanej aplikácie si uvedieme nejaké základné metriky zdrojového kódu projektu BeeeOn formou tabuľky 6.1.

	počet tried	počet .h súborov	počet .cpp súborov	počet riadkov kódu (.h)	počet riadkov kódu (.cpp)	riadkov celkom
Gateway	135	123	120	7557	16189	23746
Base	155	115	103	9105	9286	18391
Server	321	270	266	15527	27246	42773

Tabuľka 6.1: Metriky zdrojového kódu podprojektov BeeeOn.

Z tabuľky 6.1 môžeme vidieť, že projekt je pomerne rozsiahly so zdrojovými kódmi s desiatkami tísic riadkov. V súčte sa dostávame na hodnotu blízku hranici sto tisíc.

Pomerne vysoký počet tried naznačuje, že projekt je implementovaný s dôrazom na objektovo orientované programovanie. Počet tried vo všetkých troch repozitároch približne odpovedá počtu hlavičkových súborov, dokonca ich mierne prevyšuje, čo znamená, že priemerný počet definovaných tried na hlavičkový súbor je vyšší než jedna.

6.3 Metriky testov

Táto časť je venovaná metrikám samotných testov z hľadiska počtu, časovej náročnosti a pokrytia zdrojového kódu. Pre pokrytie zdrojového kódu som použil profilovací nástroj *gcov*[1].

V tabuľke 6.2 vidíme metriky integračných testov brány.

testovacia sada	počet testov	priemerná časová náročnosť
core	78	545 s
gws	17	240 s
vdev	7	82 s
psdev	1	29 s
belkin	3	186 s
celkovo	106	1085 s

Tabuľka 6.2: Metriky testovania aplikácie brány

Pokrytie kódu aplikácie po vykonaní testov vykazuje 55.36%. Na prvý pohľad sa jedná o dosť nízke číslo, no musíme pri tom uvážiť, že v súčasnosti nie sme schopní testovať všetkých správcov zariadení, kôli absencii emulátorov.

Pri hlbšej analýze kódu, ktorý aktívne testujeme, zistíme, že je pokrytie kódu pomerne vysoké. Pre príklad uvediem napríklad testovaný tlakový senzor, ktorý sme jedným komplexným testom pokryli na 87.50%. Pri bližšej analýze zistíme, že nepokrytú časť tvoria *asserty*, kontrolné podmienky a bloky ošetrujúce výnimky ktoré je pomerne náročné neinvazívne vyvolať z vonkajšku aplikácie, pretože by za normálnych okolností nemali nastať.

testovacia sada	počet testov	priemerná dĺžka trvania
gws	28	284 s
restui	81	715 s
xmlui	55	491 s
celkovo	164	1490 s

Tabuľka 6.3: Metriky testovania aplikácie BeeeOn server.

Pri integračných testoch serveru sme sa dostali na úroveň pokrytia 57.51%. Po hlbšej analýze zistíme, že je tu podobná situácia, ako v prípade pokrytia kódu brány. Hlavné použité komponenty vykazujú vysoké pokrytie. Naopak priemerné pokrytie znižuje kód tried, ktorých inštancie ani nie sú štandardne vytvorené pomocou vkladania závislostí.

Na záver vidíme v tabuľke 6.4 časovú náročnosť jednotlivých systémových testov.

	priemerná dĺžka trvania
registrácia	14 s
export	34 s
párovanie	29 s
nastavenie hodnoty	24 s

Tabuľka 6.4: Časová náročnosť systémových testov.

[1] https://gcc.gnu.org/onlinedocs/gcc/Invoking-Gcov.html

6.4 Rozšírenie práce

Modulárnosť testovacieho softvéru umožňuje rozšíriteľnosť o ďalšie testovacie moduly, ktoré by mohli zabezpečiť verifikáciu novopoužitých technológií integrovaných do systému. Integračné testy brány by sa teda mohli rozšíriť o tieto testovacie sady:

- Bluetooth – V súčasnosti aplikácia brány podporuje technológiu Bluetooth pre detekciu takýchto zariadení, čo je možné využiť pre detegovanie prítomnosti používateľa. Vhodou emuláciou takýchto zariadení by bolo možné túto podporu automaticky otestovať. Pre emuláciu by mohol byť použitý napríklad projekt *BT-sim* [2].
- Z-Wave – Ďalšou podporovanou technológiou je podpora protokolu Z-Wave, ktorá by mohla byť tiež automaticky testovaná vytvorením vhodného emulátora. Pre tento účel by mohol byť použitý projekt *py-zwave-emulator* [3].

Vhodné by bolo tiež zvýšiť celkové pokrytie kódu pre testovanie zásahom do štruktúrálnej konfigurácie aplikácie. Takéto testovanie by muselo byť úzko udržiavané a flexibilné spoločné s vývojom.

Použitý systém Kontinuálnej integrácie by sme mohli povýšiť na systém Kontinuálneho nasadenia, takže by po začlenení nejakej časti do hlavnej vetvy prebehlo automatické nasadenie systému do reálneho prostredia, čo by zefektívnilo aj akceptačné alfa testovanie.

[2] http://btsim.sourceforge.net/btsim_faq.html
[3] https://github.com/Nico0084/py-zwave-emulator

Kapitola 7

Záver

Hlavným cieľom tejto práce bolo zjednodušiť vývoj projektu BeeeOn, priamo podporiť jeho vývoj, zlepšiť kvalitu vyvýjaného softvéru a hlavne odľahčiť samotných vývojárov od náhodne sa objavujúcich regresných problémov a nutnosti riešiť softvérové chyby v pokročilejšom štádiu vývoja. Systémy *Internetu vecí* majú v dnešnej dobe čoraz väčší význam, automatizácia a monitorovanie častí bežnej domácnosti sú na vzostupe. Preto by malo byt zaisťovanie kvality týchto systémov priorita softvérových vývojárov a testerov. Tomuto účelu bola venovaná aj táto práca.

V teoretickej časti práce som sa venoval problematike softvérového testovania a automatizačných nástrojov pre priebežnú integráciu vyvýjaného softvéru. Taktiež som priblížil špecifikáciu, architekúru a ideu projektu BeeeOn ako implementácia systému *Internetu Vecí*.

V jadre práce som navrhol modulárne viacúrovňové testovanie, ktoré sa mi podarilo implementovať v jazyku Python 3 s pomocou knižnice unittest. Riešenie som integroval do systému Kontinuálnej integrácie Jenkins CI, čím je testovanie všetkých úrovní plne automatické.

Výsledok práce sú automatické testy, ktoré sú reprezentované 274 funkcionálnymi integračnými testovacími scenármi s vyše 55% pokrytím zdrojového kódu projektu, ktorý celkovo obsahuje takmer sto tisíc riadkov. Napriek regresnému zameraniu toto testovanie odhalilo počas práce viaceré závažné softvérové chyby, ktoré boli buď nahlásené a budú skoro opravené, alebo boli rovno opravené.

Literatúra

[1] *BeeeOn Github repository*. [Online; navštíveno 10.02.2018].
URL https://github.com/BeeeOn

[2] *Cppcheck*. [Online; navštívené 10.02.2018].
URL http://cppcheck.sourceforge.net/

[3] *Dependency Injection design pattern*. [Online; navštíveno 11.05.2018].
URL http://w3sdesign.com/?gr=u01&ugr=struct

[4] *Inversion of Control Containers and Dependencty Injection pattern*. [Online; navštíveno 11.05.2018].
URL https://www.martinfowler.com/articles/injection.html

[5] *Python testing.postgresql*. [Online; navštívené 10.04.2018].
URL https://pypi.org/project/testing.postgresql/

[6] *Python unittest*. [Online; navštívené 10.04.2018].
URL https://docs.python.org/3/library/unittest.html

[7] Ammann, P.; Offutt, J.: *Introduction to Software Testing*. Cambridge University Press, 2008, ISBN 0521880386.

[8] Dr. Richard Turner: *Toward Agile Systems Engineering Processes*. [Online; navštíveno 15.05.2018].
URL http://static1.1.sqspcdn.com/static/f/702523/9242881/1288744595590/200704-Turner.pdf?token=IjQArm3HfRjbuu0uOIcedtHBugY%3D

[9] Fielding, Roy Thomas: *Architectural Styles and the Design of Network-based Software Architectures*. 2000.

[10] Friedemann Mattern and Christian Floerkemeier: *From the Internet of Computers to the Internet of Things*. [Online; navštíveno 03.05.2018].
URL http://www.vs.inf.ethz.ch/publ/papers/Internet-of-things.pdf

[11] HALAJ, Jozef: *Server pro sběr senzorických dat a řízení aktivních prvků*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Viktorin Jan.

[12] Mauro Pezzè, M. Y.: *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley, 2007, ISBN 978-0-471-45593-6.

[13] Patton, R.: *Testování softwaru*. Computer Press, 2002, ISBN 80-7226-636-5.

[14] Seemann, M.: *Dependency Injection in .NET*. Manning Publications, 2011, ISBN 1935182501.
URL https://www.amazon.com/Dependency-Injection-NET-Mark-Seemann/dp/1935182501?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1935182501

[15] Shahin, M.; Babar, M. A.; Zhu, L.: *"Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices", IEEE Access, 2017*. [Online; navštíveno 11.01.2018].
URL https://arxiv.org/pdf/1703.07019.pdf

[16] Sommerville, I.: *Software Engineering (International computer science series)*. Addison Wesley, 1989, ISBN 0-201-17568-1.

[17] Sommerville, I.: *Softwarové inženýrství*. Computer Press, 2013, ISBN 978-80-251-3826-7.

Príloha A

Obsah priloženého pamäťového média

- **testing-lib** – Repozitár s pomocným kódom pre testovanie.
- **system-testing** – Repozitár scenárov systémového testovania.
- **gateway** – Repozitár BeeeOn Gateway s implementovanými testovacími prípadmi.
- **server** – Repozitár BeeeOn Server s implementovanými testovacími prípadmi.
- **doc** – Zdrojové kódy textu tejto práce.
- **DP_Richard_Wolfert.pdf** – Text tejto práce vo formáte pdf.